ORSAY
No D'ORDRE :

UNIVERSITÉ DE PARIS-SUD
U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE

présentée

pour obtenir

Le GRADE de DOCTEUR EN SCIENCES

DE L'UNIVERSITÉ PARIS XI ORSAY

PAR

Thierry GRANDPIERRE

SUJET :

Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés

Soutenue le 30 Novembre 2000 devant la commission d'examen

Alain Mérigot	Président
Michel Auguin	Rapporteur
Ahmed Amine Jerraya	Rapporteur
Anne-Marie Déplanche	Examinateur
Yves Sorel	Directeur de thèse

Table des matières

Remerciements

Je tiens à remercier sincèrement Yves Sorel pour l'ensemble de son travail d'encadrement, ses conseils, ses corrections et la confiance qu'il a su m'accorder tout au long de cette thèse.

Je remercie également Monsieur Auguin et Monsieur Jerraya qui ont accepté la lourde charge d'être rapporteurs de cette thèse, ainsi que Madame Anne-Marie Déplanche et Monsieur Alain Mérigot qui me font l'honneur de faire partie de mon Jury.

Je remercie toute l'équipe SynDEx et plus particulièrement Remy Kocik, Christophe Lavarenne et Annie Vicard qui m'ont aidé à réaliser ce travail par leurs nombreux conseils, idées et commentaires toujours constructifs.

Que tous les "bat'ouzards", de SOSSO à SAPHIR Control en passant par METALAU, trouvent ici l'expression de ma reconnaissance pour leur aide, soutien et encouragements mais aussi pour la bonne ambiance qui règne au bâtiment 12. Je tiens à exprimer toute ma reconnaissance à Martine Verneuil et Chantal Chazelas pour leur aide.

Je profite aussi de cette occasion pour remercier tous mes amis qui n'ont cessé de m'encourager. Un grand merci à Angélique Koukoutsaki et Pascal Monnier pour tous les servives qu'ils ont pu me rendre, le temps qu'ils m'ont accordé sans compter.

Enfin, je remercie tout particulièrement ma famille et ma femme, Karine, pour leur infinie patience, leurs encouragements constants, leur soutien sans faille et toute l'aide concrète qu'ils n'ont eu de cesse de me prodiguer.

A mon grand père

Introduction

Contexte

Les applications temps réel occupent une place de plus en plus importante dans le monde qui nous entoure. Longtemps réservées aux équipements industriels lourds (centrale nucléaire, chaîne de fabrication, avionique, systèmes d'armes), on les trouve maintenant dans des produits grand public (automobile, téléphone, domotique, équipements hifi et vidéo) pour lesquels les temps de développement conditionnant la mise sur le marché doivent être minimisés. Ces applications sont composées de deux parties qui interagissent. La première correspond à un système informatique lui même composé d'un calculateur qui exécute un ensemble de programmes. Ces derniers forment le *logiciel* du calculateur, ils renferment les *algorithmes* de l'application. La seconde partie d'une application correspond à son environnement physique, dont les changements d'état, sont perçus par le calculateur au moyen de capteurs. Le calculateur réagit à ces stimuli pour maintenir l'environnement dans un état déterminé au moyen d'actionneurs. Ces applications sont qualifiées de réactives puisqu'elles sont en constante interaction avec l'environnement. De plus, elles sont aussi qualifiées de temps réel car elles doivent réagir dans un temps borné afin d'assurer que l'environnement est bien contrôlé, le dépassement de ces bornes risquant d'aboutir à des conséquences catastrophiques. Enfin, la plupart de ces applications (automobiles, avions, téléphone etc) sont aussi embarquées dans le sens où on les trouve dans des équipements qui sont mobiles et pour lesquels les aspects encombrement, poids, consommation doivent être pris en compte.

Les applications temps réel reposent principalement sur des algorithmes de traitement du signal et des images et de contrôle commande qui nécessitent d'importante quantités de calculs. Lorsqu'ils doivent être effectués en un temps court, il faut utiliser des calculateurs de fortes puissances. Les limites technologiques et le coût des calculateurs puissants basés sur un unique processeur, rendent souvent impossible leur utilisation. Il faut alors utiliser des calculateurs multiprocesseur basés sur le fonctionnement en parallèle de plusieurs processeurs. Dans le cadre des applications embarquées, il est souvent plus intéressant de distribuer géographiquement les processeurs du calculateur plutôt que d'utiliser un calculateur multiprocesseur centralisé. D'une part cela permet de minimiser le câblage entre les capteurs/actionneurs et les processeurs, afin de réduire l'effet des rayonnements électromagnétiques, et d'autre part cela contribue à mieux utiliser l'espace physique disponible dans l'équipement mobile. Enfin, les calculateurs utilisés sont qualifiés d'hétérogènes car le type des processeurs et des média de communication qui les composent, sont souvent différents.

Objectifs

Cette thèse s'inscrit dans le cadre des recherches menées dans le projet SOSSO de l'INRIA-Rocquencourt, plus précisément sur la méthodologie Adéquation Algorithme Architecture (AAA) pour les systèmes informatiques temps réel distribués embarqués hétérogènes. Ces méthodes doivent aider le concepteur à atteindre les différents objectifs spécifiques au développement de ces applications.

Tout d'abord, il faut décider du nombre et des caractéristiques de chaque processeur et média de com-

munication, de façon à garantir le respect des contraintes temps réel tout en minimisant le coût et la taille de l'architecture matérielle. Ces choix ne sont pas indépendants des algorithmes de l'application qu'il s'agit d'implanter. Ainsi le logiciel doit être conçu de façon à utiliser au mieux, i.e. de manière optimisée, les ressources offertes par le calculateur distribué embarqué. D'autre part, pour minimiser le cycle de développement d'une application il faut éviter d'une part toute rupture entre la phase de spécification et celle de réalisation, et d'autre part la rupture entre le prototypage rapide de l'application et la réalisation industrielle de série. Enfin, les éventuelles conséquences catastrophiques dues au non respect des contraintes temps réel nécessitent une spécification et une réalisation la plus sûre possible, pour laquelle l'exécution du code doit être déterministe.

Afin de satisfaire toutes les contraintes que nous venons d'exposer, nous proposons une méthodologie globale (niveau système) prenant en compte toutes les phases du développement, de la spécification haut niveau des algorithmes et des architectures matérielles, à l'exécution du code. La spécification repose sur des modèles et l'exécution du code repose sur des techniques d'ordonnancement hors-ligne.

Modèles

Notre méthodologie repose sur une modélisation unifiée des algorithmes et des architectures hétérogènes distribuées spécifiés par deux graphes. Le graphe d'algorithme permet de spécifier le parallélisme potentiel que renferme l'algorithme, et le graphe d'architecture permet de spécifier le parallélisme disponible (qu'elle offre effectivement). L'implantation optimisée que nous appelons adéquation, de l'algorithme sur l'architecture, est à nouveau un graphe obtenu par transformation de ces deux graphes. Le formalisme utilisé permet de vérifier que toutes les transformations conservent les propriétés d'ordre partiel du graphe d'algorithme initial. Parmi toutes les transformations possibles, nous conservons celle qui minimise la durée d'exécution de l'algorithme. Pour cela nous utilisons une heuristique d'optimisation fondée sur de la prédiction de performances. Après avoir construit le graphe d'implantation optimisé, il est possible à l'aide d'un ensemble de règles, de générer automatiquement le programme optimisé de chaque processeur de l'architecture. Chaque programme est constitué d'une partie applicative qui correspond à l'algorithme, et d'une autre partie que nous nommons exécutif, qui gère les ressources matérielles. L'exécutif que nous construisons est généré sur mesure pour l'application de façon à minimiser les ressources qu'il lui alloue ; il repose sur un ordonnancement hors-ligne non préemptif.

Ordonnancement hors-ligne non préemptif

Il existe deux grandes catégories de politiques d'ordonnancement[17] pour répondre au problème d'allocation de ressources : d'une part les ordonnancements dynamiques qui correspondent à des politiques en-ligne, préemptifs ou non, et hors-ligne préemptif, et d'autre part les ordonnancements statiques qui correspondent à des politiques hors-ligne non préemptifs.

– Les politiques d'ordonnancement dynamiques permettent d'implanter toutes les catégories d'applications. Notons qu'elles sont plus efficaces que les politiques statiques lorsque la durée d'exécution des calculs est fortement liée à la valeur des données de ces calculs. Elles sont en apparence plus faciles à mettre en œuvre que les politiques statiques, car les exécutifs dans lesquels ces politiques sont implantées fournissent généralement un grand nombre de services qui facilitent le travail du concepteur. Cependant, ces services induisent de nombreux surcoûts, en espace mémoire (ses exécutifs sont résidants) et en temps d'exécution (changements de contextes, routage dynamique des communications) [5, 1, 68]. En utilisant ce type de politique il est plus difficile de garantir le comportement temps réel des applications. Il est donc nécessaire d'introduire d'importantes marges de sécurité conduisant finalement à gaspiller les ressources (ces marges s'ajoutant aux ressources consommées par l'exécutif).

– Les politiques d'ordonnancement statiques permettent de minimiser la durée d'exécution des applications car elles induisent de très faibles surcoûts d'exécutions [5, 32]. De part leur nature statique, il est beaucoup plus facile de prédire et de garantir le comportement temps réel des applications. Cependant leur champs d'application est plus restreint puisqu'elles requièrent, dès la compilation, de connaître toutes les caractéristiques d'une application, i.e. le système informatique et son environnement.

L'ordonnancement statique (hors-ligne non préemptif) est bien adapté [68] à l'implantation d'applications embarquées réactives de contrôle/commande, de traitement du signal et de traitement des images. En effet, comme le requiert ce type d'ordonnancement, les caractéristiques de ces applications sont connues dès la compilation puisqu'on conçoit conjointement les algoritmes et l'architecture matérielle sur laquelle ils seront implantés. Grâce à ce type de politique, il est possible de modéliser, prédire et garantir le comportement temps réel des applications. C'est pourquoi dans notre méthodologie, nous avons choisi d'ordonnancer statiquement l'exécution des calculs et des communications inter-processeurs. Ces dernières ont un impact important sur les performances des applications, de nombreux travaux s'attachent donc à les minimiser [58, 104, 67]. Cependant nous notons [32] qu'ils s'intéressent rarement à les optimiser par un ordonnancement statique car, pour cela, il faut être capable d'allouer les séquenceurs de communications qui sont rarement modélisés. Négliger les communications dans les ordonnancements statiques peut entraîner de fortes et imprédictibles dégradations de performances [1]. La modélisation de l'architecture et les techniques de générations d'exécutifs distribués taillés sur mesure que nous présentons dans cette thèse permettront d'effectuer les optimisations requises, tout en garantissant le respect des contraintes temps réel et la sûreté de spécification et de réalisation.

Plan de la thèse

Cette thèse est composée de trois parties. La première,intitulée "Spécification, implantation et optimisation" présente les trois modèles qui définissent un système temps réel distribué hétérogène, ainsi que les techniques d'optimisation que nous utilisons. Le premier chapitre concerne la modélisation des architectures distribuées hétérogènes. Après avoir réalisé un état de l'art des différents modèles d'architecture, nous présentons un nouveau modèle, suffisamment précis pour permettre la prédiction de performance et la génération automatique d'exécutif, mais suffisamment générique pour modéliser le plus grand nombre possible d'architectures avec différents types de processeurs et différents types de communications interprocesseur (communications par mémoires partagées et communications par passages de messages). Ce modèle, basé sur les graphes orientés, permet de modéliser les séquenceurs de calculs et de communications des processeurs ainsi que les différents types de mémoires et bus, interconnectant ces éléments. Dans le second chapitre, nous définissons le modèle d'algorithme. C'est un hypergraphe orienté d'opérations, infiniment itéré et conditionné, auquel est associé un ordre partiel d'exécution sur les opérations. Il décrit la partie réactive de l'application. Le troisième chapitre définit le modèle d'implantation qui représente l'ensemble des implantations possibles d'un algorithme sur une architecture à l'aide de la composition de trois relations : le routage, la distribution et l'ordonnancement. Le routage permet de rendre complètement connecté une architecture si elle ne l'est pas, la distribution partitionne le graphe d'algorithme en fonction du graphe d'architecture ; à chaque élément de partition est associé un ordre partiel. Enfin l'ordonnancement construit un ordre total pour chaque élément de partition créé par la distribution. Le quatrième chapitre décrit comment, parmi l'ensemble des implantations, on sélectionne celle qui minimise la durée d'exécution de l'application ainsi que la mémoire de l'architecture. Pour cela ont utilise une heuristique d'optimisation basée sur les dates d'exécutions des calculs et les quantités de mémoires manipulées par ces calculs.

La seconde partie de cette thèse, intitulée "Génération automatique d'exécutifs distribués", décrit toutes les étapes qui, à partir d'un graphe d'implantation optimisé, permettent de générer un exécutif distribué optimisé taillé sur mesure pour l'application. Dans le cinquième chapitre, après avoir réalisé un état de

l'art des différents types d'exécutifs existants, nous justifions notre choix de générer automatiquement un exécutif statique taillé sur mesure. Le sixième chapitre décrit et valide à l'aide des réseaux de Pétri, les transformations du graphe d'implantation optimisé en un graphe d'exécution sur lequel ont été ajoutées des opérations systèmes (contrôle de séquencement, synchronisation, initialisation) nécessaires à l'exécution de l'application par les processeurs de l'architecture. Dans le septième chapitre, nous donnons les règles qui permettent de transformer un graphe d'exécution en un macro-code intermédiaire. L'usage d'un macro-code intermédiaire associé à des bibliothèques génériques nous permet d'utiliser des techniques de générations d'exécutifs indépendantes des langages utilisés par les compilateurs des processeurs. Le huitième et dernier chapitre de cette seconde partie, décrit la structure des bibliothèques génériques d'exécutifs que nous utilisons, ainsi que le processus de transformation du macro-exécutif généré automatiquement en un exécutif compilable taillé sur mesure pour l'application.

La troisième partie de cette thèse présente les développements logiciel effectués, ainsi que deux exemples d'application de la méthodologie AAA. Ainsi, après avoir effectué un état de l'art des différents outils logiciel du domaine dans le neuvième chapitre, et avoir constaté l'inexistence de logiciels adaptés aux besoins que nous avons spécifié, nous présentons dans le dixième chapitre le logiciel SynDEx qui implante cette méthodologie. A partir de la spécification d'un graphe d'algorithme et d'architecture et de leurs caractéristiques, SynDEx effectue automatiquement la distribution et l'ordonnancement des calculs et des communications et l'affiche sous la forme d'un diagramme temporel. Si la contrainte temps réel est satisfaisante, SynDEx est capable de générer automatiquement l'exécutif spécifique à l'application. Nous présentons la structure de ce logiciel qui a été redéveloppée en C++ et Tcl/Tk dans le cadre de cette thèse, afin d'y apporter de nouvelles fonctionnalités. Le cœur de ce logiciel a fait l'objet d'une spécification détaillée basée sur le formalisme OMT orienté objet. Le onzième chapitre donne deux exemples d'utilisation de la méthodologie pour des applications concrètes tirées du monde industriel : le Cycab, un véhicule électrique semi-autonome, et le projet PROMPT du Réseau Nationnal de Recherche en Télécommunications.

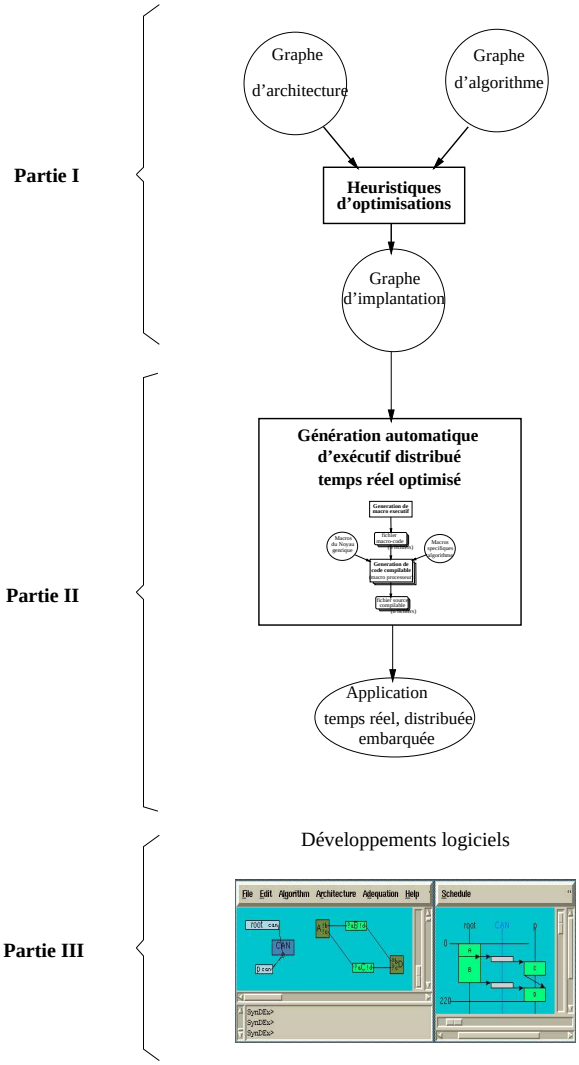


FIG. 1: *Plan de la thèse*

Première partie

Spécification, implantation et optimisation

Chapitre 1

Modèle d'architecture

Sommaire

L'objectif de la méthodologie AAA, présentée dans cette thèse, est d'implanter rapidement mais de façon optimisée un algorithme sur une architecture multiprocesseur distribuée embarquée. Le mot *implantation* doit être ici pris au sens large. Il inclut non seulement la programmation par traduction à l'aide d'un langage de l'algorithme en du code machine, mais aussi tout le processus de développement qui mène à cette étape ultime. Cela comprend la spécification de l'application (algorithme et architecture matérielle), la distribution

et l'ordonnancement optimisé des composants de l'algorithme sur les composants de l'architecture, puis la génération automatique de code, son chargement et son exécution dans chaque composant de l'architecture.

Dans l'introduction, nous avons vu que les optimisations, dans la conception des systèmes distribués temps réel embarqués, consistent à minimiser la latence de l'application de façon à garantir le respect des contraintes temps réel et à minimiser la taille de l'architecture (en utilisant au mieux les ressources matérielles disponibles). Pour pouvoir optimiser ainsi, il est nécessaire de disposer d'un modèle qui reflète précisément le comportement de l'architecture (les optimisations sont en partie effectuées à partir de prédictions de performances), tout en étant suffisamment adapté à la génération automatique de code. L'automatisation de cette étape doit permettre de garantir que toutes les optimisations effectuées sont conservées lors de l'exécution réelle, elle doit aussi permettre d'accélérer le prototypage qui pourra alors être qualifié de "rapide".

Dans ce chapitre nous commencerons par faire un état de l'art des machines parallèles, préalable nécessaire à l'étude des différents modèles existants. Après avoir montré l'absence de modèle adapté à nos objectifs, nous présenterons et justifierons notre modèle d'architecture de la méthodologie AAA, basé sur des graphes orientés.

1.1 État de l'art

En 1946, von Neumann[1] a posé les bases d'un modèle d'architecture encore largement utilisé aujourd'hui. Ce modèle repose sur la coopération de cinq unités (Cf. figure 1.1). L'*unit´e m´emoire* contient des instructions et des données. L'*unit´e de traitements* transforme (effectue des calculs) les données stockées en mémoire. Les *unit´es d'entr´ee et de sortie* permettent de transférer des données entre la mémoire et l'environnement. L'unité de commande contrôle l'ensemble de ces unités, elle repose sur un *s´equenceur d'instructions* qui lit ses instructions dans la mémoire et les applique à l'unité de traitement. L'ensemble unité de commande-unité de traitement est souvent appelé *processeur* ou CPU (Central Processing Unit).

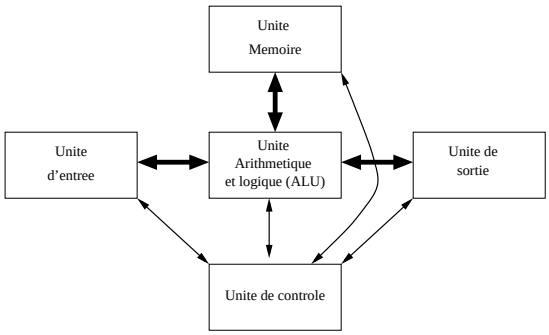


FIG. 1.1: *Modèle de Von Neumann (les flèches en gras représentent les chemins de données, les flèches fines les chemins de contrôle)*

1.1.1 Classification des processeurs

Bien que toujours basés sur le principe de von Neumann, les processeurs ont bénéficié de nombreuses modifications architecturales afin d'améliorer leur puissance de calcul. Il est ainsi possible de les classer selon leur architecture interne [47].

1. avec l'aide de Eckert et Mauchly

1.1.1.1 Processeurs CISC

L'unité de traitement d'un processeur était initialement capable de réaliser un petit nombre d'opérations arithmétiques et logiques (ALU). Pour accroître les performances des processeurs, les concepteurs ont modifié l'unité de contrôle et l'unité de traitement de façon à pouvoir supporter des opérations et des modes d'adressage (calcul d'adresses mémoire à partir de valeurs de registres) de plus en plus complexes, d'où le nom de **C***omplex* **I***nstruction* **S***et* **C***omputer*. Le séquenceur d'instructions ne pouvant plus être uniquement cablé, étant donné sa complexité [48], a rapidement nécessité un fonctionnement basé sur des micro-programmes[2]. Basé sur ce fonctionnement complexe, les instructions complexes nécessitent en moyenne une dizaine de cycles d'horloge.

1.1.1.2 Processeurs RISC

Dans les années 1980 [84, 82], il a été montré que les processeurs CISC passaient 80 pour cent de leur temps à exécuter 20 pour cent du jeu d'instructions offert et que que la réduction [82] du nombre d'instructions (**R***educed* **I***nstruction* **S***et* **C***omputer* [49]) accompagnée d'une simplification de leur format (taille fixe des instructions, emplacement fixe des "codes opérations" et des opérandes) permettait de simplifier leur décodage et de le câbler complètement (sans utilisation de micro-programme). Ainsi, en apportant simplicité et régularité il a été possible de pipeliner le décodage et l'exécution des instructions, permettant finalement de réduire leur durée d'exécution.

Souvent couplés à une architecture HARVARD (ref harvard) (dans laquelle les instructions et les données sont lues dans des mémoires caches ou des mémoires physiques différentes, connectées au processeur par des bus séparés) les processeurs RISC permettent d'exécuter une instruction par cycle d'horloge.

L'utilisation efficace de ces processeurs repose en grande partie sur la capacité de leurs compilateurs à bien gérer les registres internes. Le travail qui était effectué par les micro-programmes des processeurs CISC est effectué en grande partie par les compilateurs des processeurs RISC.

1.1.1.3 Processeurs VLIW

Ces processeurs sont basés sur un séquenceur d'instructions et plusieurs unités de traitement qui peuvent travailler en parallèle pour effectuer des opérations très différentes sur des données différentes. Pour cela, chaque instruction du processeur contient une instruction spécifique pour chaque unité de traitement. Les instructions de ces processeurs sont donc stockées dans des *mots* de grande longueur, d'où leur nom de **V***ery* **L***ong* **I***nstruction* **W***ord* [107]. Ces processeurs offrent un parallélisme de calcul qualifié de *parall´elisme à grain fi n* car il est observé au niveau des instructions du processeur. Ce parallélisme est si fin que c'est essentiellement sur les compilateurs que repose l'exploitation des unités parallèles [30]. Ces processeurs sont rarement classés parmi les *machines SIMD* [53] (présentées dans la section qui suit), car bien que n'ayant qu'un seul séquenceur d'instructions, les opérations effectuées par les unités de traitement ne sont pas nécessairement identiques comme c'est le cas pour les machines SIMD dont la structure est régulière.

1.1.1.4 Processeurs de Traitement du Signal - DSP

Dans le but d'accélérer les calculs spécifiques au domaine du traitement du signal (filtrage, fft), un autre type de processeur a été créé. Il s'agit des processeurs de traitement du signal (DSP - Digital Signal Processor), ils possèdent pour la plupart [71, 47] :

- des unités capable(s) d'effectuer en parallèle la lecture de deux opérandes, une multiplication, et une

2. Les instructions complexes correspondent `a des micro-programmes stock´es dans une m´emoire interne qui avait l'int´erˆet d'ˆetre plus rapide que le reste de la m´emoire et permettait d'utiliser des registres de donn´ees interm´ediaires tr`es rapides eux aussi

opération arithmétique (cela permet d'accélérer par exemple la somme de produits - MAC *Multiply ACcumulate* - utilisée dans le produit de matrices pour le calculs de filtres),

- des modes d'adressage spécialisés (par exemple pré et post modification des pointeurs d'adresse, adressage circulaire, adressage *bit-reverse*, pour accélérer le calcul des FFT),
- des instructions de contrôles particulières (grâce à une architecture adaptée) qui permettent l'exécution de boucles sans surcoût temporel (i.e. sans gaspiller de cycle à incrémenter, puis tester le compteur de boucles et à effectuer le saut conditionnel arrière vers le début de la boucle),
- l'architecture des DSP est souvent faite de façon à permettre plusieurs accès simultanés à la mémoire pendant l'exécution d'une instruction arithmétique. Ainsi ils ont souvent de la mémoire interne pour les données et une mémoire cache instruction pour faire de "l'optimisation 1 cycle": lecture 2 opérandes, calcul, écriture dans le même cycle.

Ces processeurs offrent donc un parallélisme à grain fin qui ne peut être exploité que par la connaissance précise et spécifique de l'architecture du processeur. Comme pour les processeurs RISC, les performances de ces processeurs dépendent fortement de la capacité de leurs compilateurs à bien placer les données [71].

1.1.1.5 Microcontrôleurs

Basés sur les modèles de processeurs CISC ou RISC, ils sont conçus pour fonctionner avec un minimum de composants extérieurs. Ainsi leur mémoire est souvent embarquée sur le chip ainsi que de nombreuses unités d'entrée-sortie (CAN [106], UART RS232, I2C [80], pour échanger des données avec des périphériques (claviers, afficheurs) et processeurs, PWM [3] pour piloter des moteurs, convertisseurs analogique/numérique ou numérique/analogique pour appréhender le monde physique extérieur . . .). En contrepartie leur puissance de calcul est souvent relativement faible étant donné la surface de silicium restante et les contraintes d'embarquabilité et de consommation minimale (puisque la fréquence de fonctionnement est souvent diminuée). Dans le contexte des systèmes embarqués, ils sont destinés à être implantés géographiquement près des capteurs et actionneurs, où ils réalisent alors souvent des calculs simples liés à l'acquisition ou à la commande.

1.1.1.6 Les Systèmes "On Chip" (SOC)

Avec l'émergence des SOCs (Système sur un même morceau de silicium, "System On a Chip"), qui incluent diverses unités (processeurs, mémoires, systèmes d'entrée-sortie), la définition d'un processeur devient plus floue : par exemple, le TMS320C80 [98] de Texas Instrument est qualifié de "processeur" alors qu'il inclut quatre processeurs DSPs, un processeur RISC, cinq bancs mémoire, un crossbar et un DMA sur un seul chip. Nous donnerons notre définition de processeur dans la prochaine section.

1.1.2 Classification des machines

A partir d'un seul processeur il est possible de construire des machines relativement simples, de puissance égale à celle du processeur, accroître la puissance de calcul de la machine équivaut à accroître celle du processeur. Une autre technique que celle de l'utilisation de processeur unique, consiste à construire des *machines parallèles* par connexion de processeurs à l'aide de mémoires et de média de communication. Dans ces machines, les unités de traitement de chaque processeur fonctionnent en parallèle de façon à résoudre un problème commun. Pour cela ils doivent coopérer, c'est à dire communiquer pour s'échanger des

3. Pulse Width Modulated

données mais aussi se synchroniser. Il est possible de connecter ensemble de telles machines pour en créer de plus puissantes encore. Ainsi, nous constatons que toutes les machines qu'il est possible de construire, sont basées sur trois notions génériques : le *traitement* (basé sur le séquencement d'instructions réalisant des opérations arithmétiques et logiques), *la m´emoire* et la *communication*. Nous allons utiliser ces trois notions pour classer les machines. La première identifie le type de *parall´elisme* offert par la machine, il est basé sur la relation entre le nombre de séquenceurs et le nombre d'unités de traitements arithmétiques et logiques. Le second critère repose sur l'organisation de la mémoire dans la machine et le troisième identifie le type de communication qui peut avoir lieu dans la machine.

1.1.2.1 Séquenceurs

Dans la célèbre classification de Flynn [33], qui a maintenant 30 ans, les machines sont organisées en quatre groupes correspondant au parallélisme d'instruction (un ou plusieurs séquenceurs qui permettent un ou plusieurs flux d'instructions) et de données (une ou plusieurs unités de traitement qui permettent un ou plusieurs flux de données).

Les *machines SISD* sont basées sur un unique séquenceur d'instructions et une unique unité de traitement. Ce sont les machines les plus simples, elles n'offrent aucun parallélisme, leur architecture est comparable à celle de von Neumann. Les *machines SIMD* n'ont toujours qu'un séquenceur d'instructions mais renferment plusieurs unités de traitement fonctionnant en parallèle de manière *synchrone*, multipliant d'autant leur capacité de calcul. Les *machines MISD* sont des machines possédant plusieurs séquenceurs d'instructions mais une seule unité de traitement (il n'existe pas encore de machine commerciale de ce type [49] particulier, les machines pipeline sont parfois classées dans cette catégorie). La dernière catégorie, très en vogue ces dernières années grâce aux progrès technologiques, correspond aux *machines MIMD*. Elles possèdent plusieurs séquenceurs d'instructions indépendants et plusieurs unités de traitements et fonctionnent de ce fait de manière principalement *asynchrone*. Les MIMD sont *homogènes* si les séquenceurs et les unités de traitements qui les composent sont identiques, et *h´et´erogènes* si ils sont différents. Il existe une très grande diversité de machines MIMD aux caractéristiques architecturales très différentes (selon par exemple, l'organisation de leurs mémoires ou de leurs modes de communication), montrant ainsi la limite de la classification de Flynn.

1.1.2.2 Mémoire(s)

L'organisation de la mémoire et les méthodes d'accès à la mémoire par les processeurs sont la base d'une seconde classification [9] qui organise les machines MIMD en trois catégories : UMA, NUMA et NORMA.

Dans les machines de type UMA (*Uniform Memory Access*), tous les processeurs ont accès à toute la mémoire de la machine de façon uniforme. C'est le cas des machines à mémoire centralisée et partagée (*Centralized Shared Memory*) entre tous les processeurs au moyen d'un bus. Ce bus devient rapidement le goulet d'étranglement lorsque les requêtes des processeurs à la mémoire sont nombreuses et simultanées, il engendre donc un phénomène de contentions des processeurs qui est proportionnel à leur nombre dans l'architecture. Ce nombre est donc généralement limité, selon la technologie employée, à moins d'une trentaine de processeurs [48]. Pour tenter de résoudre ce problème, deux types de variante architecturale ont été introduites. La première repose sur l'ajout de mémoire cache au niveau de chaque processeur. La seconde consiste à diviser la mémoire en bancs physiques distincts. Chaque banc peut être mis en relation avec n'importe quel processeur au moyen d'un réseau d'interconnexion dynamique (capable de supporter de forts débits et des temps d'accès très courts : bus hiérarchisés, multiétage, crossbar). Bien que physiquement découpée, la mémoire reste logiquement continue et accessible de façon uniforme par tous les processeurs qui peuvent aussi être connectés à une mémoire cache pour diminuer davantage la latence d'accès à la mémoire.

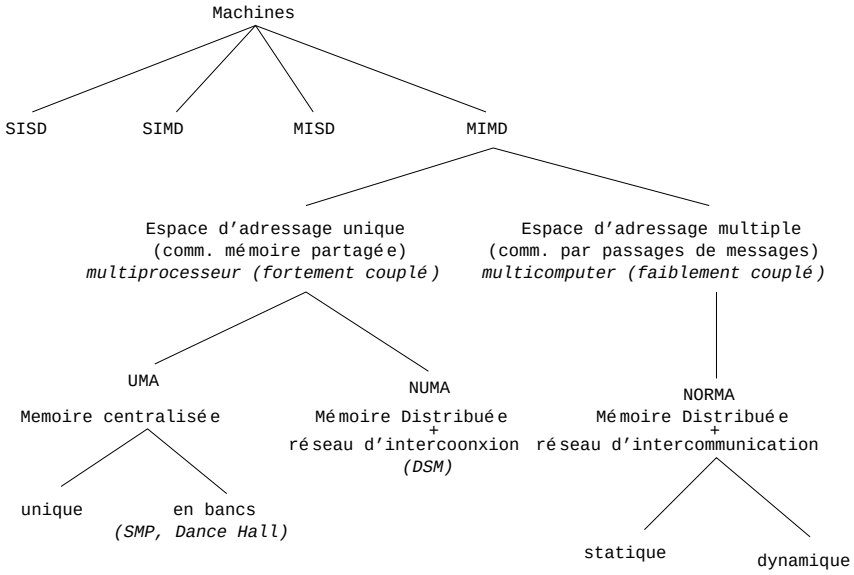


FIG. 1.2: *Classification des différents types de machines*

Comme l'architecture des machines UMA est très symétrique et composée de plusieurs processeurs on les appelle parfois SMP (*Symmetric MultiProcessor*) ou *"dance hall"* [4] [107].

Dans les machines de type NUMA (*Non Uniform Memory Access*), tous les processeurs ont accès à tout l'espace mémoire de la machine (comme dans les machines UMA), mais le temps d'accès n'est plus uniforme, il dépend de la localisation géographique des données dans la machine. C'est le cas des machines à mémoire distribuée-partagée (*Distributed Shared Memory* - DSM). Chaque processeur est connecté directement à une mémoire locale et à un bus ou à un réseau d'interconnexion qui lui permet d'accéder à chaque mémoire locale de chaque processeur de la machine (l'accès à sa mémoire locale étant plus rapide puisque ne passant pas par le réseau partagé, l'accès n'est plus uniforme). Bien que physiquement distribuée dans toute la machine, la mémoire est toujours vue, par tous les processeurs, comme un espace unique logiquement continu).

Tous les processeurs des machines de type UMA et NUMA se partagent le même réseau d'interconnexion, même pour lire leurs instructions, c'est pourquoi on dit que ces architectures sont *fortement coupl´ees* . Le réseau d'interconnexion, par sa bande passante, limite le nombre de processeurs qu'il est possible de connecter, ne permettant pas d'étendre facilement le nombre de processeurs. Ces architectures sont donc difficilement extensibles mais sont simples à programmer puisque la mémoire est accessible par tous les processeurs (des optimisations plus complexes sur le placement des données doivent cependant être effectuées pour minimiser les transferts de données des machines NUMA). Les deux types de machines appartiennent à la catégorie des *multiprocesseur.*

Dans les machines de type NORMA (*NO Remote Memory Access*), les processeurs n'ont plus accès à toute la mémoire de la machine. Cela correspond aux machines à mémoire physiquement distribuée (comme les NUMA), mais non partagée. Chaque processeur est connecté à une mémoire locale (qui lui est privée), et il ne peut accéder à la mémoire locale d'un autre processeur. Les échanges de données se font de manière explicite sur un *r´eseau de communication* qui connecte les processeurs. Comme chaque processeur est connecté à sa mémoire, il peut fonctionner quasi indépendamment des autres, ils appartiennent donc à la catégorie des *multicomputer,* ce sont des machines *faiblement coupl´ees.* Elles ont l'intérêt d'être facilement

4. par analogie avec les salles de danse o`u fi lles et gar¸cons sont situ´es de part et d'autre du hall

extensibles (i.e. extension du nombre de processeurs qui la compose) mais sont plus difficiles à programmer puisque le programmateur doit transférer explicitement les données à communiquer.

1.1.2.3 Communications

Nous avons vu que les communications au sein des machines multiprocesseur (UMA et NUMA), et des machines multicomputer (NORMA), ne reposaient pas sur les même principes (la figure 1.2 récapitule leurs relations).

RAM Dans le cas des premières machines, les communications sont dites par *m´emoire partag´ee* (*shared memory*) car tous les processeurs ont accès à toute la mémoire, qu'elle soit physiquement centralisée ou distribuée. Les communications sont implicites dans ce type de machine puisqu'il n'est pas nécessaire pour le programmeur de distinguer les données qui sont échangées entre les processeurs de celles qui ne le sont pas. Cette mémoire est toujours appelée RAM (Random Access Memory) pour deux raisons :

- quand la mémoire est de dimension N (c'est à dire composée de N registres correspondant à N emplacement spatiaux différents), il est possible d'accéder aléatoirement, en lecture ou écriture, à n'importe lequel d'entre eux car ils ont tous une adresse différente,
- l'ordre de lecture des données est totalement indépendant de leur ordre d'écriture, l'accès est aléatoire. Il est possible d'écrire plusieurs fois un même registre sans avoir lu sa valeur entre temps, on peut avoir $nW \rightarrow 1R$, où W est une écriture, R une lecture et n quelconque.

SAM Dans les machines NORMA, où la mémoire est distribuée mais non partagée, les communications se font de manière explicite par *passage de messages* (*message passing*) sur le réseau d'intercommunication. Dans ce type de machine le programmeur doit distinguer les données à échanger entre les processeurs. Le réseau d'intercommunication peut être basé sur de la mémoire RAM partagée ou de la mémoire SAM (Sequential Access Memory) [41]. Cette dernière, définie comme une extension du modèle SAM de [46], possède deux caractéristiques qui la distingue des mémoires RAM :

- ce type de mémoire impose que chaque écriture dans la mémoire soit suivie d'une lecture (i.e. il ne peut y avoir deux écritures successives si il n'y a pas eu une lecture entre les deux), les données sont lues dans l'ordre de leur écriture. Cela correspond physiquement aux liaisons par mémoire FIFO, aux liaisons parallèles, séries point-à-point ou multipoint. Si la mémoire SAM est composée de N registres (on parle souvent de FIFO à N places), et en reprenant la notation introduite dans le paragraphe précédent, on a maintenant $nW \rightarrow mR$ avec $N \geq n \geq m$. En effet, après avoir écrit n données dans la FIFO, il est possible de n'en lire que $m \leq n$. Lorsque la FIFO est pleine, il faut au moins faire une lecture avant de pouvoir faire une écriture, la lecture est impossible quand la FIFO est vide. La FIFO que nous évoquons ici, est souvent physiquement distribuée à chaque extrémité du média physique de communication (lien de DSP TMS 320C40 ou ADSP21060, bus CAN, bus ethernet etc) : lorsque la FIFO d'émission est pleine, les données sont transférées dans la FIFO distante, si il y reste de la place,
- quelque soit le nombre de registres de cette mémoire, elle ne possède qu'une adresse, les données sont donc toujours lues ou écrites à la même adresse. Les données stockées dans la mémoire SAM sont référencées temporellement par leur ordre d'écriture et non spatialement par une adresse comme dans le cas des RAM.

Les mémoires SAM point-à-point ne peuvent être connectées qu'à deux processeurs. Chaque écriture dans la mémoire par un processeur doit être suivie d'une lecture par l'autre processeur. Si la mémoire est

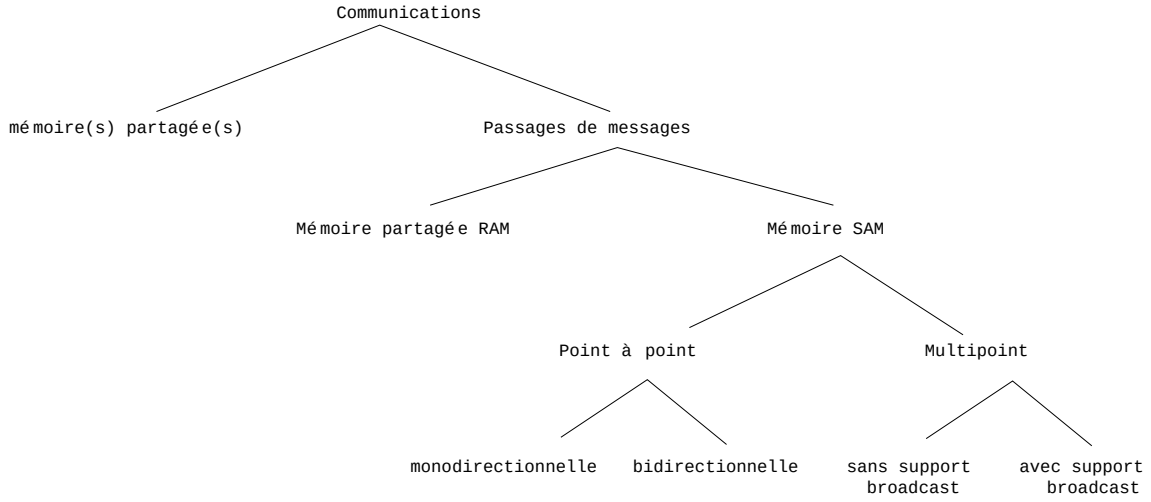


FIG. 1.3: *Les différents type de mémoire*

monodirectionnelle, un seul des processeurs est capable d'écrire dans la mémoire, l'autre étant uniquement capable d'y lire. Si elle est bidirectionnelle, les deux processeurs connectés sont capables d'y écrire (mais toute écriture de l'un doit être suivie d'une lecture par l'autre).

Les mémoires SAM multipoint peuvent être connectées à plus de deux processeurs. Il existe deux types de mémoire SAM multipoint selon qu'elles supportent ou non la diffusion matérielle. Lorsque la mémoire supporte la diffusion matérielle - que nous appellerons toujours *broadcast* par la suite - tous les processeurs connectés peuvent faire une lecture de la même donnée simultanément. Si elles ne supportent pas le broadcast, un seul des processeurs est capable de lire une donnée. Si une donnée doit être utilisée par plusieurs processeurs, il faut effectuer autant d'écritures qu'il y a de destinataires. La figure 1.3 présente les différents types de mémoires et leurs variantes. Le tableau suivant récapitule les différences entre les mémoires RAM et les mémoires SAM.

	Mémoire			
	RAM		SAM	
	Nombre de registres :		Nombre de registres :	
	1	N	1	N
Ordre entre R/W	$nW \rightarrow 1R$ aléatoire	$nW \rightarrow 1R$ aléatoire	$1W \rightarrow 1R$ fixe	$nW \rightarrow mR$ fixe (avec $N \geq n \geq m$)
Adressage spatial	fixe (adress. uniq.)	aléatoire (adress. multip.)	fixe (adress. uniq)	fixe (adress. uniq.)

Remarque 1 *Il existe des architectures hybrides dans lesquelles les processeurs peuvent communiquer par mémoire partagée mais aussi par passage de messages. Certaines d'entre elles sont organisées en grappes (cluster). Chaque grappe est basée sur un petit nombre de processeurs (un multiprocesseur) dans lequel les communications peuvent se faire par mémoire partagée. Les communications entre grappes se font par passages de messages.*

La classification proposée par Jonhson [54], propose d'organiser les machines MIMD en fonction de leur structure mémoire et de leurs communications. Le tableau suivant illustre cette classification dans laquelle

nous avons précisé les correspondances avec les méthodes d'accès à la mémoire (UMA,NUMA,NORMA) et les types de mémoire (RAM ou SAM) des média de communication :

		Communications	
		Shared Variables (RAM)	Message Passing (RAM/SAM)
Memory Structure	Global Memory	GMSV (UMA)	GMMP (NORMA)
	Distributed Memory	DMSV (NUMA)	DMMP (NORMA)

1.1.2.3.1 Unité DMA Les mémoires partagées, qu'elles soient SAM ou RAM, ont généralement une bande passante inférieure à celle du processeur et des autres RAM. Lorsqu'un processeur effectue un transfert de données entre deux mémoires, il passe donc une partie de son temps à attendre que les données soient prêtes du fait de la différence de bande passante. Pour ne pas gaspiller ainsi leur temps, les processeurs sont presque toujours dotés d'un mécanisme d'interruption.

Par exemple, lorsqu'une mémoire SAM est pleine, celui-ci est parfois capable de le signaler en émettant un signal d'interruption (signal FIFO pleine). Lorsque le processeur reçoit ce signal, il interrompt (ou *pr´e-empte*) les opérations de calcul en cours afin d'exécuter des instructions appropriées à la gestion de la SAM. Le processeur peut ensuite reprendre l'exécution des opérations de calcul interrompues. Ce mécanisme n'apporte cependant pas de réel parallélisme entre les calculs et les communications, il permet d'éviter au processeur de perdre de précieux cycles à attendre que les données soient dans la mémoire qui devrait être scrutée en permanence (cela correspond à du "polling" actif de la mémoire).

Pour libérer en partie le processeur de la gestion fine des transferts de données entre la mémoire et les unités d'entrées-sorties, une unité DMA est parfois ajoutée aux processeurs. Cette unité permet un réel parallélisme entre calculs et communications car elle est capable d'accéder directement au contenu de la mémoire (*Direct Memory Access*) pour transférer des données contigües en mémoire depuis et vers les unités d'entrée-sortie. Cette unité possède son propre séquenceur de transfert, partiellement indépendant du séquenceur d'instruction du processeur et la rend partiellement autonome. L'indépendance n'est que partielle car un DMA n'est généralement pas capable de lire ses instructions en mémoire, il requiert le séquenceur d'instruction pour être programmé et lancé. Les fonctionnalités des DMA ont été largement étendues dans les machines MIMD pour permettre de transférer directement des données entre les mémoires de processeurs différents à travers le réseau d'intercommunication (basé sur de la mémoire SAM ou RAM), offrant ainsi un réel parallélisme entre calculs et communications.

Le DMA est souvent constitué de plusieurs canaux qui correspondent chacun à un ensemble de registres et permettent d'effectuer plusieurs transferts en parallèle. La programmation d'un canal du DMA consiste à charger ses registres, qui sont au minimum au nombre de quatre : un registre d'adresse source de la zone à transférer, un registre pour l'adresse de destination, un registre pour le nombre de mots à transférer et éventuellement un registre pour la valeur de l'incrément des adresses. A cela s'ajoute un registre d'état, non programmable, qui fournit l'état courant du DMA, et un registre de contrôle. C'est à travers ce dernier registre qu'est lancée l'exécution du transfert par le DMA.

Une fois programmé et déclenché, le DMA génère/séquence des adresses et des signaux de contrôle (lecture/écriture) ce qui permet le transfert des données entre les deux zones (source et destination) spécifiées, sans intervention du séquenceur d'instructions qui peut donc effectuer d'autres opérations pendant ce temps. Ces zones mémoires peuvent être internes, externes, des FIFOs de lien de communication, d'entrée-sortie. L'automate du DMA effectue automatiquement les opérations suivantes (schématisées par la figure 1.4) :

1. Transfert (adresse source → adresse destination),

2. Incrémente adresse(s) d'une quantité égale au pas,
3. Décrémente compteur,
4. Si compteur différent de 0 retourne en 1,
5. Déclenche éventuellement une interruption de fin de transfert.

(à chaque étape le registre d'état du DMA est mis à jour)

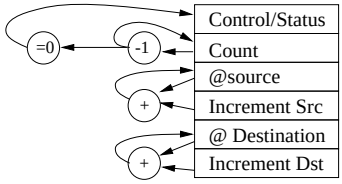


FIG. 1.4: *Automate du DMA*

Quand un transfert est achevé, le DMA signale au séquenceur qu'il a fini et qu'il faut le reprogrammer pour un nouveau transfert en émettant un signal d'interruption spécifique. Ainsi, si l'interruption du DMA est plus prioritaire que le traitement en cours d'exécution, le séquenceur d'instruction sauvegarde son contexte (pointeur programme et les registres utilisés) et exécute une procédure de traitement associée à cette interruption (c'est en général à travers elle que les registres du DMA sont re-programmés). Quand la fin de la routine d'interruption est atteinte, le processeur restaure le contexte sauvegardé pour poursuivre l'exécution du programme interrompu.

Remarque 2 *Certains DMA (comme celui du TMS320C40 et son "link pointer") sont capables, lorsqu'ils ont achevé un transfert, d'aller lire une zone de la mémoire dans laquelle se trouvent les paramètres d'un autre transfert (source, destination etc), il peut donc s'auto-re-programmer pour effectuer un nouveau transfert et enchaîner ainsi plusieurs transferts sans requérir le séquenceur d'instruction. Cependant, cette reprogrammation automatique est inconditionnelle. Le DMA ne sait pas exécuter d'instruction de rupture de séquence pour faire du saut conditionnel ce qui est indispensable pour se synchroniser avec le séquenceur d'instruction de calcul. Dans ce cas il n'est pas possible de se synchroniser avec le séquenceur d'instruction sans que celui-ci prenne le contrôle total du DMA.*

Remarque 3 *Les DMA disposent en général de plusieurs canaux ("Canaux DMA"). Ils sont ainsi capables de transférer différents blocs de données entre différentes mémoires quasi-simultanément en interlaçant les transferts de façons à profiter de la latence d'accès à la mémoire.*

1.1.2.3.2 Structure du réseau La structure du réseau peut être *statique* ou *dynamique* (multiétage) [22]. Dans un réseau statique, les interconnexions sont déterminées par construction matérielle (p17 de [87]), chaque processeur peut communiquer directement avec un nombre fixe et déterminé de processeurs. Les performances des machines reposant sur ces types de réseaux sont fortement dépendantes de la distribution des données et des calculs sur chaque processeur. Les machines idéales sont donc complètement connectées, mais cela est peu réaliste (avec la technologie actuelle) lorsque le nombre de processeurs est grand. Les réseaux *reconfigurables* appartiennent aussi à cette catégorie car les connexions n'y sont pas modifiables pendant l'exécution des programmes. Les réseaux dynamiques reposent sur des *commutateurs*, appelés aussi *routeurs*. Souvent associées aux processeurs, ces unités sont dédiées aux routages des communications dans le réseau.

Le *routage* [89] consiste à rechercher le chemin à prendre dans un réseau pour transmettre des messages entre deux processeurs. Les *routeurs*, couplés à des DMA sont des unités dédiées à la gestion de la transmission des messages dans le réseau, ce sont des commutateurs programmables munis d'une logique de commande plus ou moins complexe (Cf. paragraphe suivant). Ils permettent de libérer le processeur de l'analyse et du routage des messages qui transitent sur le réseau. Le routage doit permettre de minimiser la durée des communications entre les processeurs en minimisant les conflits dans le réseau.

Modes de commutation Sur les réseaux statiques non complètement connectés, les communications et leur routage doivent être effectués explicitement par le programmeur, ou effectués de façon logicielle par le système d'exploitation et/ou des librairies (PVM [37], MPI [92] par exemple). Il repose donc souvent sur un mécanisme simple de *store and forward* dans lequel toutes les données sont transmise et stockées de proche en proche. Les routeurs des réseaux dynamiques déchargent le programmeur de la gestion des communications, grâce à différentes techniques de commutation souvent implantées matériellement :

- dans la *commutation de circuit*, avant d'envoyer son message, la source envoie une requête jusqu'à la destination afin de construire un circuit physique de bout-en-bout. Une fois le circuit établi, le message est alors transmis directement jusqu'à destination,
- dans *la commutation de paquets* (un paquet est un ensemble de données souvent étiqueté par sa destination, son origine, sa taille) chaque routeur dispose de tampons mémoires, chacun pouvant stocker un paquet. Lors de la réception d'un paquet, le routeur le stocke dans un des tampons avant de le retransmettre au routeur suivant sur la route du paquet. Chaque paquet contient des informations permettant au routeur de calculer le chemin à suivre. Ce principe repose donc sur le store and forward vu précédemment,
- dans le mécanisme *"Whormole"*, le message est découpé en petites entités appelées *flits* qui sont stockées dans les tampons mémoires des routeurs. Seul le premier flit contient des informations permettant au routeur de calculer le chemin à suivre, tandis que les autres flits ne contiennent que des données. Ils doivent donc avancer les uns à la suite des autres dans le réseau.

1.1.2.3.3 Topologie La topologie d'un réseau d'interconnexion correspond à sa structure matérielle, à la façon dont sont connectés les processeurs (ou les routeurs) entre eux. Elle est souvent dictée par le type des algorithmes, ou imposée par l'environnement. En effet dans les systèmes embarqués, les capteurs et les actionneurs, connectés aux unités d'entrée-sortie des processeurs, sont souvent physiquement distribués. Pour minimiser les câblages, les processeurs sont souvent placés près de ces capteurs et actionneurs, ce qui conduit à des topologies d'interconnexion parfois irrégulières. Le classement des différents types de topologie repose sur trois critères [53], la *distance*, *le diamètre* et la *connectivit´e*. Les topologies les plus répandues reposent sur les bus (simples, multiples, hiérarchisés), les réseaux à connexions directes (totalement connectés, anneau, étoile, grille, hypercube) ou hiérarchisés (arbres binaires, arbre anneau, pyramide).

1.1.2.4 Définitions

Après ces trois types de classement (par séquenceur, mémoire et communication), il est maintenant possible d'établir quelques définitions génériques de machines.

Machines distribuées, ou réparties Les architectures parallèles faiblement couplées (NORMA) sont généralement qualifiées de machines (parfois systèmes) distribuées ou réparties. Ces machines sont obtenues par connections de machines qui ont la propriété d'être spatialement distantes les une des autres plutôt que

centralisées dans un unique lieu physique. Dans le cadre des systèmes temps réel embarqués, cet éloignement est souvent engendré par la nécessité technique de rapprocher les processeurs des capteurs et actionneurs pour minimiser les câblages des signaux analogiques.

La différence entre les qualificatifs "distribué" et "réparti" est souvent floue et propre à chaque domaine d'application. Cette différence n'existe pas en anglais puisque le mot réparti n'a pas d'équivalent direct dans cette langue.

Par exemple, la littérature française consacrée aux systèmes transactionnels qualifie souvent la machine de répartie par extension de l'utilisation qui est faite des données par les applications qui y sont implantées. En effet, dans les systèmes transactionnels les données sont réparties (bases de données réparties []). Chaque processeur de l'architecture exécute un programme indépendamment des programmes exécutés sur les autres processeurs, ces programmes peuvent coopérer sans qu'il soit possible de reconstituer une horloge ou un temps global. Pour un observateur extérieur, les processeurs communiquent de façon aléatoire.

Une machine répartie est parfois distinguée d'une machine distribuée [26] par l'introduction d'une notion de site répartiteur principal (mais non central) chargé de faire la répartition. Cette notion n'existe pas dans le cas des machines distribuées.

Dans la littérature consacrée aux systèmes embarqués, par exemple de l'automobile, l'architecture parallèle est parfois dite *distribu´ee* relativement à la démarche adoptée lors de sa programmation ainsi que du fonctionnement qui en résulte. Cela correspond à une méthodologie de conception "bottom-up" où l'architecture distribuée est obtenue par connection de processeurs ayant des rôles prédéfinis dès la conception de l'architecture. Chaque calcul est donc contraint sur un processeur (on parle aussi de partitionnement fonctionnel), souvent sans tenir compte du problème global d'équilibrage de charge de l'architecture Il n'est pas possible de reconstruire une horloge globale pour ce type d'architecture. Au contraire, l'architecture est dite *r´epartie* quand sa programmation a suivie une méthode globale "top-down" où l'architecture est considérée dans son ensemble durant toute la phase de développement, cela permet par exemple de mieux prendre en compte les problèmes d'équilibrage de charge ou d'optimisation globale.

Par la suite, et afin d'éviter toute confusion, nous utiliserons uniquement le terme machine distribuée pour qualifier l'architecture matérielle parallèle du système informatique faiblement couplé. La méthodologie AAA est basée sur une approche globale "top down", sur chaque processeur de ces machines nous allons *distribuer* et *ordonnancer* les calculs de l'algorithme de façon à faire différentes optimisations (mémoire, durée d'exécution, taille de l'architecture . . .).

Machine parallèle homogène, hétérogène La machine parallèle est dite homogène quand toutes les machines qui la composent sont identiques (i.e. leurs processeurs sont de même type) et que le réseau de communication est basé sur des *composants identiques* (les composants représentant les mémoires RAM ou SAM, les DMA étudiés précédemment). C'est le cas des SMP, de la plupart des machines UMA (puisque l'accès à la mémoire doit être uniforme) et plus généralement de la plupart des machines fortement couplées qui nécessitent de communiquer de façon homogène. Les machines faiblement couplées, c'est à dire les machines distribuées selon la définition donnée dans le paragraphe précédent, peuvent être homogènes ou hétérogènes, ainsi que leurs réseaux de communications.

Machine synchrones et asynchrones Une machine fonctionne en mode synchrone lorsque tous les séquenceurs travaillent de manière synchrone sous le contrôle d'une horloge physique commune. Dans le mode asynchrone, qui est celui de toutes les machines faiblement couplées (NORMA) et aussi la plupart des machines fortement couplées (UMA et NUMA), chaque séquenceur travaille sous une horloge spécifique. La cohérence de l'exécution distribuée de l'algorithme doit être garantie par le programmeur qui doit insérer des points de synchronisation appropriés chaque fois que cela est nécessaire. Une variante (Cf. BSP

§ 1.1.3.1) consiste à utiliser des barrières de synchronisation : tous les processeurs collaborent pour réaliser une tâche commune, lorsque chaque processeur a fini sa partie, il se met en attente. Lorsque tous les processeurs sont en attente, la barrière de synchronisation peut être franchie.

Dans cette thèse nous traiterons des machines distribuées qui sont donc asynchrones. Nous proposerons des techniques de synchronisation, indispensables à l'exécution distribuée des algorithmes, dans le chapitre *Ordonnancement* et dans la partie *g´en´eration automatique d'ex´ecutif distribu´e.*

1.1.3 Les modèles existants

Pour décrire le fonctionnement et la programmation des machines séquentielles, le modèle le plus employé reste celui de von Neumann puisqu'il modélise correctement les machines SISD actuelles. Dans le monde des machines distribuées et parallèles, aucun modèle ne s'est vraiment imposé pour faciliter leur programmation. Dans cette section nous avons classé les modèles existant en trois grandes catégories : les modèles de haut niveau (correspondant à un haut degré d'abstraction de la machine), les modèles de bas niveau (très proches de la réalité), et divers modèles implicites et non formalisés que nous avons rencontré dans un certain nombre d'outils logiciel.

1.1.3.1 Modèles de Haut niveau

Dans cette section nous présentons des modèles dont la plupart sont classés parmi les "modèles de calcul parallèle" dans [50].

PRAM (machine a mémoire partagée)

Le modèle PRAM [34] est un des premiers modèles de haut niveau développé en 1970 pour programmer les machines parallèles. Dans ce modèle la machine est vue comme un ensemble de processeurs séquentiels indépendants et communiquant par une mémoire globale partagée [22] c'est à dire :

- p processeurs P_0 à P_{p-1},
- m positions mémoire M_0 à M_{m-1}.

Ce modèle est basé sur trois règles fondamentales :

1. un programme peut connaître le contenu d'une mémoire M_i avec "Lire(M_i)", et le modifier avec "Écrire(M_i)",
2. la séquence suivante d'opérations est atomique (elle prend une unité indivisible de temps pour s'exécuter) :
   - Lire(M_i)
   - Calculer f
   - Écrire(M_i)
3. Les opérations sont exécutées de façon synchrone : au même instant tous les processeurs lisent, calculent et écrivent.

Ce modèle est très utile pour dégager le parallélisme des problèmes étudiés. Il constitue souvent une première étape pour la parallélisation. Étant donné son haut niveau d'abstraction, il permet souvent de savoir si un problème peut être parallélisé ou non et dans quelle mesure. La description des algorithmes est très simple car il suffit de décrire une séquence d'opérations parallèles exécutée par les processeurs sans se préoccuper des communications entre les processeurs. La plupart des questions théoriques s'expriment naturellement dans ce modèle.

L'inconvénient est qu'il est fortement éloigné des machines réelles. La plupart des machines multiprocesseur ne sont pas à accès uniforme car les contraintes technologiques permettant à un grand nombre de processeurs d'accéder en temps constant à une mémoire commune sont telles qu'aucune machine PRAM n'a encore vu le jour. Par conséquent il faut souvent réadapter l'algorithme PRAM à la structure de la machine choisie, ce qui est d'autant plus complexe si la machine communique par passage de messages, ce qui induit un perte de temps de développement non négligeable.

Ce modèle reste néanmoins d'actualité et un certain nombre de variantes a été proposé, par exemple en modifiant le principe d'accès à la mémoire :

- lecture Exclusive (*ER*) : un processeur peut lire une case mémoire à un instant donné,
- lecture Concurrente (*CR*) : plusieurs processeurs peuvent lire une même case mémoire à un instant donné,
- écriture Exclusive (*EW*) : un processeur peut écrire dans une case mémoire à un instant donné,
- écriture Concurrente (*CW*): plusieurs processeurs peuvent lire une même case mémoire à un instant donné. Pour résoudre les conflits trois modèles de politique ont été exposés :
  1. modèle *commun* : tous les processeurs doivent écrire la même valeur dans la même case,
  2. modèle *arbitraire* : un processeur quelconque réussit à écrire, mais l'algorithme doit toujours s'exécuter correctement,
  3. modèle *prioritaire* : ordre de priorité sur les processeurs, c'est celui qui a la plus haute priorité qui écrit.

Il est possible de classer ces variantes par ordre de complexité de réalisation :
$EREW, CREW, CRCW_{commun}, CRCW_{arbitraire}, CRCW_{prioritaire}$

Intérêt des PRAMs Ce modèle ignore la complexité de l'algorithme sur la connectivité (le programmeur n'a pas à tenir compte des communications, synchronisations, localisation des données), le programmeur peut se focaliser sur les problèmes de calculs et non d'implantation. De nombreux algorithmes efficaces sont conçus sur ce modèle. Il a donné naissance à de nombreux paradigmes de design très robustes ayant des applications sortant des PRAMs. Les dernières avancées ont démontré que les algorithmes étaient formellement émulables sur des machines hautement connectées.

Reproches au modèle PRAM Ce modèle ne tient pas compte de la latence d'accès à la mémoire qui existe dans les machines réelles, les résultats obtenus peuvent ainsi être décevants une fois implantés sur les machines existantes. La variante CRCW est inutilisable car CR et CW sont difficilement réalisables (impossible d'atteindre la vitesse plus élevée de la version CRCW sur une version EREW). De plus il n'aide pas à la gestion des communications et synchronisations puisque cette partie n'est pas modélisée mais idéalisée. Ce modèle convient aux machines SIMD, mais il ne prend pas en compte les temps de communication, et n'est pas directement applicable aux ordinateurs parallèles basés sur des réseaux complètement connectés (bien

qu'une machine UMA puisse simuler un PRAM). Enfin, ce modèle ne fournit qu'une évaluation qualitative des performances.

Remarque 4 *Les deux premiers points sont contournables en introduisant des commutateurs pour réaliser des machines basées sur CR et CW, les autres points sont plus difficiles à satisfaire facilement. Ce modèle est quand même très pratique pour la conception d'algorithmes parallèles utilisant des structures de données très différentes et pour l'analyse des perfectionnements des algorithmes indépendamment des machines réelles [107].*

Il existe deux extensions du modèle permettant de modéliser la programmation d'une architecture plus réaliste :

Weakly Coherent PRAM : WPRAM La WPRAM [76] repose sur un espace d'adressage partagé faiblement cohérent nécessitant que les processus participent à la synchronisation si ils ont de nouvelles données à partager.

Broadcast with Selective Reduction : BSR La BSR [99] est basée sur N processeurs se partageant M places mémoires. Tous les processeurs peuvent accéder à tous les emplacements mémoires au même instant (instruction broadcast) pour écrire et pour chaque place mémoire on choisit un sous-ensemble de données qui est réduit à une valeur. Son implantation est aussi rapide que les CRCW PRAM et peut s'effectuer par circuit combinatoire, elle n'est pas plus complexe que EREW PRAM. Elle peut être implantée par un :

- bus mémoire et arbre combinatoire,
- "mesh of trees" : chaque processeur est connecté à la racine d'un arbre dont les extrémités sont connectées à la mémoire,
- des circuits de tris et de calculs de préfixes (permettant un temps constant pour le broadcast).

DRAM (machine à mémoires distribuées) Cosnard et Fereira [21] ont proposé une généralisation du modèle PRAM pour prendre en compte les accès mémoire dans le cas d'une mémoire distribuée. Comme dans le cas PRAM, tous les processeurs fonctionnent de manière synchrone. Dans ce modèle, chaque processeur possède sa propre mémoire locale de taille constante et il n'existe pas de mémoire partagée. Les processeurs communiquent uniquement par un réseau d'interconnexion. Il est défini par un ensemble de p processeurs P_i, p zones mémoire M_i et d'une famille de couples $X = (i, j)$ représentant le réseau d'interconnexion des DRAM. P_i ne peut accéder qu'aux zones mémoires M_j, quelque soit j appartient X_i.

BSP, LogP, CGM

Récemment, de nombreux travaux se sont attachés à décrire des modèles prenant en compte les caractéristiques réelles des machines, tout en essayant d'englober le plus grand nombre de machines possible. Au contraire du modèle PRAM, le coût des communications n'est plus ignoré, cependant la topologie du réseau de communication n'est pas spécifiée précisément ([63]).

Ces modèles sont tous basés sur un modèle de machine correspondant à un ensemble de processeurs interconnectés par un réseau. Un processeur peut être une machine monoprocesseur (SISD), un processeur d'une machine MIMD (NUMA ou NORMA) ou une machine MIMD elle-même. Le réseau peut être n'importe quel support de communication entre les processeurs (bus, réseau, mémoire partagée). La connaissance exacte de la topologie du réseau n'est pas nécessaire pour concevoir des algorithmes dans ce modèle.

BSP Le modèle BSP (Bulk Synchronous Parallel) [101], formalise les caractéristiques architecturales des machines existantes en peu de paramètres. Il est basé sur un ensemble de processeurs séquentiels, chacun doté d'une mémoire locale et connectée aux autres par un réseau spartiate de communication basé sur le passage de messages. Chaque processeur peut ainsi faire des calculs en utilisant fréquemment des références locales et quelques références globales moins fréquentes. Dans ce modèle le temps de communication est d'ordre log p (où p = nombre de processeurs).

Un algorithme, écrit dans le modèle BSP, est constitué d'une séquence de *super-´etapes*. Lors d'une super-étape, un processeur peut faire des calculs locaux et un nombre limité de communications (émissions et/ou réceptions). Deux super-étapes consécutives sont synchronisées par une barrière de synchronisation. Le modèle BSP utilise différents paramètres pour caractériser les super-étapes, comme L la période de synchronisation qui correspond à L unités de temps nécessaires pour synchroniser tous les processeurs, g le coût pour envoyer un mot à travers le réseau, h le nombre maximal de messages que peut envoyer ou recevoir chaque processeur, ainsi que s le surcoût fixe dû à la mise en place d'une communication, aussi petit que soit le message à envoyer. Le coût des communications se calcule à l'aide de ces paramètres. Une phase de communication demande un temps $gh + s$. Le coût des calculs locaux correspond à la somme des coûts unitaires de chaque opération élémentaire.

LogP Le modèle LogP [23] apporte plus de précision par rapport à BSP en reflétant plus précisément les caractéristiques des machines réelles. Les quatre lettres du mot LogP représentent le coût des communications : L la latence, o le surcoût fixe d'une communication (temps de traitement d'un message par un processeur), g le pas c'est à dire l'intervalle de temps minimum entre deux envois ou réceptions consécutives de messages sur un processeur et P le nombre de processeurs. Ce modèle suppose que le réseau a une capacité finie telle qu'au plus L/g messages peuvent être envoyés ou reçus par un processeur à chaque étape. L, o et g sont donnés comme étant des multiples du cycle du processeur (durée d'une opération élémentaire).

Les processeurs travaillent de manière asynchrone contrairement à BSP. Les communications entre les processeurs sont de type point-à-point. LogP aide au partitionnement des données et à l'ordonnancement des tâches pour limiter les communications.

CGM Le modèle CGM (Coarse Grained Multicomputer) [25] est une simplification de BSP. Ce modèle s'affranchit des paramètres L,g,h et s, ainsi que de l'étape de synchronisation. Les algorithmes écrits dans ce modèle sont composés d'une succession de deux phases : une phase où chaque processeur effectue un calcul sur ses données et une phase où les processeurs échangent des données afin de les redistribuer. La phase de communication correspond à une communication globale entre tous les processeurs.

1.1.3.2 Modèles non formalisés

La plupart des modèles suivants ne sont pas présentés clairement comme tels. Ce sont des modèles implicites et sous-jacents aux principaux outils de conception (présentés en détail dans la dernière partie de cette thèse) utilisés pour programmer des machines parallèles de notre domaine. Nous avons cependant jugé nécessaire de les présenter ici brièvement puisque cela permettra de situer notre modèle.

Apotres

C'est un outil qui ne traite que les architectures SIMD. L'architecture est définie par le nombre de processeurs, leur puissance (nombre de cycle de calcul par seconde), la quantité de mémoire répartie uniformément sur l'ensemble des processeurs et la bande passante des communications (nombre de cycles

machine nécessaires à la communication d'un paquet de données entre une paire de processeurs). La durée des communications est supposée indépendante de la topologie.

CASCH

Cet outil suppose une architecture homogène, modélisée par un graphe où chaque sommet est un processeur et les arcs représentent des liens permettant des communications par passages de messages. La topologie d'interconnexion est libre, mais n'est pas utilisée pour les optimisations temporelles. Les communications reposent sur un système d'exploitation (operating system) indépendant de l'outil.

GEDAE

L'architecture y est décrite par un fichier de caractéristiques dans lequel figure la liste des processeurs, le mécanisme de communications entre les processeurs (socket, mémoire partagée, passage de messages, accès direct à la mémoire), la quantité de mémoire.

Ptolemy

L'architecture n'est pas modélisée par un graphe, elle est décrite par un module "target" (cible) dont le format n'est pas ouvert. Il existe différentes cibles : ordonnanceur monoprocesseur, fully connected (multiprocesseur complètement connecté), shared bus (processeurs connectés par un bus). Le modèle de l'architecture est donc difficilement formalisable et limité à quelques topologies particulières.

PYRROS

L'architecture est homogène et basée sur un réseau régulier (cube etc), les processeurs communiquent uniquement par passages de messages sur des liens point-à-point.

Trapper

L'architecture peut être hétérogène, elle est modélisée par un graphe (appelé graphe matériel) où les noeuds représentent les processeurs, et les arcs connectant les noeuds représentent les canaux de communications. Ce graphe peut être hétérogène puisque les processeurs peuvent être de types différents. Il est dit que Trapper supporte les communications par passage de messages et les communications par mémoires partagées. Cependant, la possibilité de faire des communications par mémoire partagée n'est pas modélisée au niveau du graphe.

1.1.3.3 Modèles de Bas niveau

Langages HDL, modèle RTL Les concepteurs de processeurs utilisent des langages spécifiques pour décrire leur processeur (*Hardware Description Language*, HDL). Les HDLs couvrent plusieurs niveaux de description. Cela peut aller de la description fonctionnelle et doit descendre jusqu'à la description physique du processeur (au niveau portes logiques, transistors). Ce sont en effet les points d'entrée des outils de synthèse de silicium et de simulation. Par exemple les langages de spécification VHDL [52, 2], MIMOLA[71], et des langages plus anciens [46] KARL (Karlsruhe Architectural and Register Transfer Langage), ABL (Architectural Blockdiagram Langage), CDL (Computer Design Langage) permettent de spécifier les architectures à plusieurs niveaux et notamment jusqu'au niveau de la combinatoire (transferts) entre les registres *RT Level* [46] (Register Transfer Level). Citons enfin le *processor level*[46], basé sur le langage KARL (donc d'un niveau d'abstraction supérieur au niveau RTL), il divise un processeur en deux parties : la partie

données et la partie contrôle. La partie données est faite de registres, d'opérateurs combinatoires et de chemins de données contrôlables qui permettent la manipulation de données. Cette partie reçoit des signaux de contrôle de l'unité de contrôle et envoie à son tour des signaux informant de son état. Nous verrons par la suite que notre modèle se situe hiérarchiquement juste au dessus de ce niveau.

Les "Block diagram" Chaque concepteur de processeurs et de cartes fournit dans les databooks des "blocks diagram" de leurs architectures. Ce sont des schémas fonctionnels de l'architecture interne des processeurs. Il n'existe pas de standard pour ce type de modélisation, le niveau de détail est variable. Ils font apparaître, pour la plupart, les unités de calculs, les bus internes, les unités de communications, les unités spécifiques ainsi que les bus qui les connectent. Cette modélisation est parfois qualifiée de "bus système" [75], l'exemple de la figure 1.5 donne l'équivalent de l'architecture de von Neumann présentée précédemment (figure 1.1), l'unité de contrôle et celle de traitement sont encapsulées dans le CPU. La modélisation d'architecture réelle sera souvent conçue à partir des informations extraites de ce type de schéma (Cf schéma des figures 1.34 et 1.40).

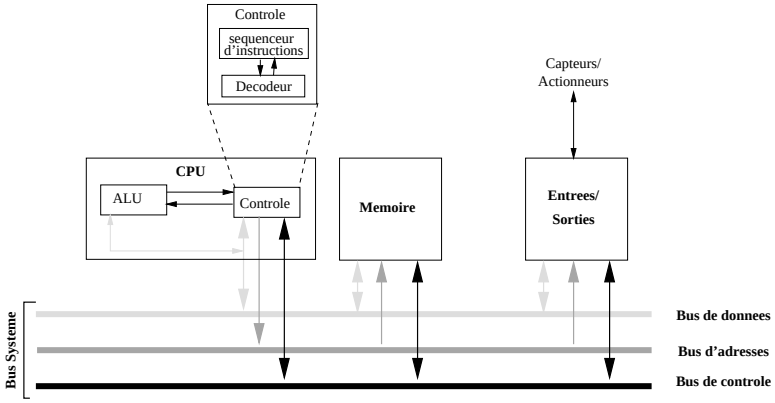


FIG. 1.5: *Les bus systèmes*

1.2 Le modèle AAA

1.2.1 Objectifs

Cet état de l'art nous a permis de bien cerner le type de machines à modéliser, mais il a aussi mis en évidence l'inadéquation entre les modèles existants et les objectifs fixés. Ainsi, les modèles que nous avons regroupé dans la catégorie des "modèles de haut niveau" conviennent peu à l'implantation des algorithmes dans le contexte du temps réel. En effet, aucun d'entre eux ne reflète fidèlement le comportement de l'architecture, notamment au niveau des communications qui sont souvent idéalisées (temps de communication ignorés PRAM DRAM ou topologie non prise en compte) et non hétérogènes. Enfin, ces modèles font souvent l'hypothèse d'un fonctionnement synchrone de la machine (PRAM, DRAM; barrière de synchronisation dans BSP, LogP; phases de calculs alternées avec phases de communications dans CGM), alors qu'elles sont plus souvent asynchrones (Cf. § 1.1.2.4) dans les machines distribuées-embarquées.

Les modèles de "bas niveau" sont les plus proches de la réalité physique de la machine, ils permettent ainsi la prédiction de performance nécessaire à la phase d'optimisation. Cependant, leur grain de descrip-

tion trop fin les rend rapidement inexploitables [5] dans le cadre du prototypage rapide de machines complexes construites avec plusieurs processeurs. La spécification trop peu générique est nécessairement longue à mettre en œuvre, cela se traduit par des temps de spécification beaucoup trop longs dans le cadre du prototypage rapide. De plus elle requiert des informations qui ne sont pas toujours disponibles auprès des concepteurs de matériel.

Les modèles sous-jacents aux outils logiciels étudiés, et surtout celui de GEDAE, semblent les plus proches de ce que nous cherchons. L'architecture y est décrite à un niveau qui encapsule les détails très fins de l'architecture, tout en ne posant pas trop d'hypothèses simplificatrices, sauf pour les communications qui sont souvent négligées. En effet, Trapper et CASCH délèguent ce problème de communication au système d'exploitation, tandis que Pyrros restreint la topologie à des réseaux très réguliers.

Finalement aucun modèle ne nous convient puisque, soit la modélisation est trop fine et entraîne une complexité d'utilisation trop élevée dans le cadre du prototypage rapide, soit la modélisation est trop grossière et ne permet pas d'effectuer les optimisations requises dans le cadre des systèmes temps réel embarquée. C'est pourquoi, dans la suite de ce chapitre nous allons présenter notre modèle d'architecture qui doit remplir les contraintes suivantes :

1. il est nécessaire de modéliser précisément toutes les ressources programmables de l'architecture puisque l'on doit générer un exécutif pour chacune d'elles. Cela inclut les séquenceurs d'instructions et de transferts et tous les types de média de communication ;

2. nous nous plaçons dans le cas d'exécutifs statiques basés sur un ordonnancement hors ligne non préemptif (Cf chapitre introduction), ce qui nous permet d'optimiser fortement l'implantation, le modèle doit être compatible avec cette politique d'implantation ;

3. l'optimisation est basée sur l'utilisation du parallélisme offert par l'architecture. Le modèle d'architecture doit donc mettre clairement en évidence le *parall´elisme effectif* de l'architecture, on parle aussi de *parall´elisme disponible*, cela inclut le parallélisme entre les calculs, mais aussi entre les calculs et les communications ;

4. l'optimisation est basée sur la prédiction de performances de la machine, elle nécessite donc de modéliser précisément l'arbitrage de toutes les ressources partagées. La qualité des optimisations obtenue est directement liée au niveau de précision dans la caractérisation de chacun des éléments de l'architecture ;

5. étant donné que la complexité du problème de distribution-ordonnancement croît exponentiellement avec le nombre de composants de l'architecture et d'opérations de l'algorithme, il est indispensable de minimiser ce nombre en faisant de "l'abstraction". Cela consiste à réduire la complexité jusqu'à un niveau "tolérable" d'approximation. La difficulté est de choisir le bon niveau d'abstraction, si il est trop élevé le résultat de la prédiction sera imprécis (donc peu optimisé) et la génération de code impossible, si il est trop faible la prédiction et le processus d'optimisation vont prendre un temps très long, voir infini à l'échelle humaine ;

6. enfin, le modèle de machines distribuées doit aussi permettre de décrire le plus grand nombre de machines existantes et à venir. La modélisation doit donc être précise pour permettre de bonnes prédictions, mais cependant générique afin de permettre d'optimiser l'utilisation du plus grand nombre d'architectures.

5. M^eme en utilisant des heuristiques pour r´esoudre le probl`eme d'allocation de ressource, le temps n´ecessaire est beaucoup trop grand si la description est trop fi ne.

En résumé, la modélisation doit être générique et rigoureuse puisqu'après l'étape d'optimisation il faudra faire générer automatiquement le code de l'application, en garantissant que ce code conserve les propriétés de la spécification algorithmique initiale (exécution sans interblocage comme nous le verrons dans la seconde partie). Nous avons donc choisi de modéliser l'architecture à l'aide de graphes orientés présentés ci-après, ensuite nous définirons les règles permettant à partir d'une description de machines distribuées parallèles hétérogènes d'en tirer un graphe la modélisant avec précision.

1.2.2 Description et justification

Nous modélisons l'architecture hétérogène distribuée par un graphe orienté (S, A), où S est l'ensemble des sommets de ce graphe et A l'ensemble de ses arcs. Chaque sommet est une *machine à ´etats fi nis* qui produit et consomme des données, chaque arc correspond à une connexion entre deux machines à états finis. L'ensemble des sommets S se décompose en 5 sous-ensembles disjoints qui correspondent chacun à un type de machine à états finis, soit :

- S_{opr} l'ensemble des opérateurs,
- S_{RAM} l'ensemble des registres, mémoires RAM et mémoires RAM partagées,
- B_{bus} l'ensemble des Bus/Mux/Demux, Bus/Mux/Demux/Arbitre-RAM et Bus/Mux/Demux/Arbitre-SAM,
- S_{SAM} l'ensemble des Mémoires SAM,
- S_{com} l'ensemble des communicateurs.

1.2.2.1 Opérateur connecté à un registre

Chaque opérateur représente un séquenceur d'opérations, la ou les unités de traitement commandées par ce séquenceur, ainsi que les unités d'entrées sorties éventuelles connectées aux capteurs et actionneurs. Si l'on définit une opération comme étant faite d'un ensemble d'instructions (Cf. Modèle d'algorithme), les opérateurs peuvent être assimilés aux séquenceurs d'instructions des machines SIMD, MIMD présentés au § 1.1.2.1. Chaque opérateur exécute séquentiellement un sous-ensemble des opérations du graphe de l'algorithme stockées dans des *registres* accessibles par l'opérateur. Un registre permet de mémoriser une donnée à travers un mécanisme d'écriture et de lecture effectué par l'opérateur qui lui est connecté, il n'impose aucun ordre entre lecture et écriture, son accès est donc qualifié d'aléatoire (Cf. § 1.1.2.3). Les opérations sont soit de type calcul, soit de type entrée-sortie. Une opération de calcul consomme des données dans un ou plusieurs registres (défini dans la section suivante) qui doivent être connectés à l'opérateur, effectue des calculs à partir de ces données, puis écrit les données produites dans un ou plusieurs registres. Le rôle d'une opération d'entrée est uniquement de produire des données à partir d'informations physiques issues de capteurs faisant partie de l'opérateur, alors que le rôle d'une opération de sortie est uniquement de consommer des données pour les appliquer à des actionneurs faisant partie de l'opérateur. Les actionneurs transforment les données en grandeurs physiques (Cf. modèle d'algorithme).

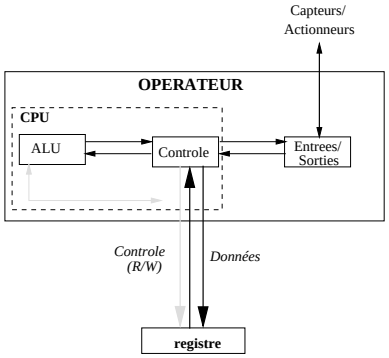


FIG. 1.6: *Opérateur connecté à un registre*

L'architecture la plus simple (Cf. figure 1.6 et modélisation AAA figure 1.7) est basée sur un opérateur *d´eg´en´er´e* connecté à un seul registre. Un opérateur dégénéré n'est capable d'exécuter qu'une seule opération qu'il n'est pas nécessaire de stocker dans un registre. L'opérateur peut lire ou écrire dans ce registre (en fonction des signaux de contrôle R/W) qui n'impose aucun ordre entre lecture et écriture. La vitesse maximum d'accès à un registre est définie par sa bande passante noté $BPmax$. Cette architecture est donc composée (Cf. § 1.7) d'un sommet opérateur et d'un sommet registre connectés par deux arcs de sens opposés puisque les données peuvent suivre les deux directions. Pour alléger la représentation des graphes d'architectures, deux arcs orientés de sens opposés connectant les mêmes sommets seront par la suite représentés par un arc non orienté.

a) graphe complet b) représentation simplifiée

FIG. 1.7: *Opérateur connecté à un registre*

Les opérateurs sont caractérisables, entre autre (nous le verrons en détail dans le chapitre caractérisation), par l'ensemble des opérations qu'ils sont capables d'exécuter, la durée d'exécution de chacune de ces opérations, le nombre d'arcs que l'on peut y connecter (Cf. exemple opérateur DSP § 1.2.2.10, et pour chaque arc une bande passante maximum).

1.2.2.2 Opérateur connecté à n registres

Un unique registre est souvent insuffisant pour implanter un algorithme, il alors nécessaire que l'opérateur puisse lire ou écrire dans des registres différents. De plus, si l'opérateur n'est pas dégénéré (i.e. si il est capable d'exécuter plusieurs opérations), il faut plusieurs registres pour stocker sa séquence d'opérations. Un opérateur n'étant généralement pas capable d'accéder simultanément à tous les registres, il doit sélectionner un registre parmi les n disponibles avant chaque lecture ou écriture. Pour cela les registres sont connectés à l'opérateur par un bus et un multiplexeur-demultiplexeur [6] (Cf. figure 1.8) commandé par un décodeur d'adresses. Chaque registre est alors associé à une adresse et accédé selon le principe suivant :

- dans le cas d'une écriture, l'opérateur génère une adresse qui, décodée par le décodeur d'adresse, configure le démultiplexeur de façon à ce que la donnée soit stockée dans le registre correspondant,

6. Le multiplexeur aiguille les donn´ees dans le sens $n \rightarrow 1$, alors que le d´emultiplexeur aiguille les donn´ees dans le sens $1 \rightarrow n$

- dans le cas d'une lecture, le processeur génère une adresse qui configure le multiplexeur de façon à ce que la donnée du registre sélectionné parmi les n soit lue par l'opérateur.

Lors d'une écriture les données traversent donc un démultiplexeur alors que pendant la lecture elles traversent un multiplexeur.

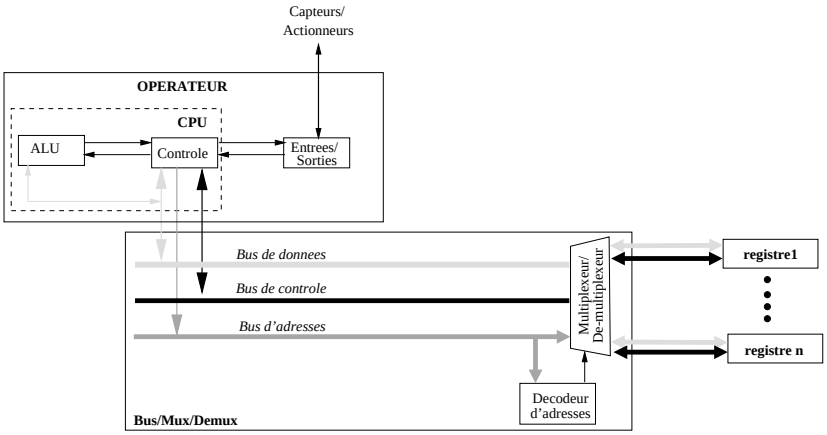


FIG. 1.8: *Opérateur capable d'accéder à un registre parmi n*

Nous encapsulons l'ensemble bus, multiplexeur, démultiplexeur et décodeur d'adresse dans un sommet unique appelé Bus/Mux/Demux. Ce sommet est connecté au sommet opérateur et à tous les registres. Les arcs qui connectent les sommets représentent explicitement le sens des transferts de données entre les sommets mais pas celui des adresses qui suit implicitement toujours la direction opérateur → registre puisque seul les opérateurs (et les communicateurs Cf. § 1.2.2.9) sont capables de générer des adresses (Cf. figure 1.9).

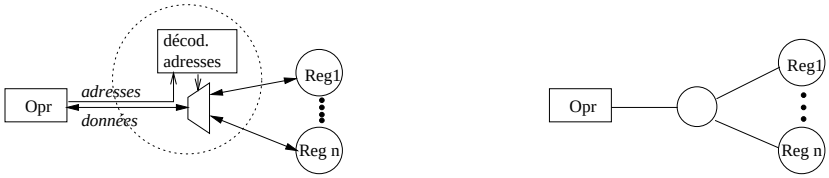


FIG. 1.9: *Opérateur connecté à n registres par un Bus/Mux/Demux*

1.2.2.3 Mémoire RAM

Si tous les registres connectés à un Bus/Mux/Demux ont des caractéristiques homogènes, il est possible d'encapsuler le Bus/Mux/Demux et tous les registres dans un unique sommet appelé mémoire RAM (Cf. figure 1.10). Sa bande passante est égale à celle des registres qui la composent. Comme indiqué en 1.1.2.3, l'accès aux registres par l'opérateur est aléatoire puisqu'il peut lire ou écrire dans des adresses quelconques, mais aussi parce que l'ordre entre lecture et écriture est aussi quelconque.

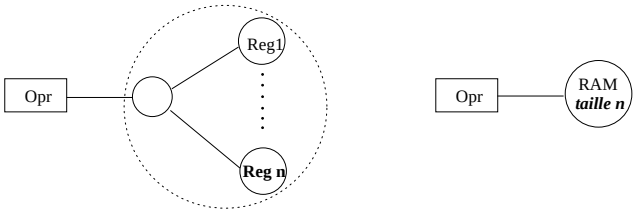


FIG. 1.10: *Sommet mémoire RAM*

On distingue trois types de mémoire RAM selon l'usage qui en est fait par l'opérateur qui lui est connecté :

- la mémoire programme est utilisée uniquement pour stocker les instructions des opérations,
- la mémoire données est utilisée uniquement pour stocker des données produites et consommées par les opérations,
- la mémoire programme et données, dite *mixte*, peut contenir à la fois les instructions des opérations et les données produites et consommées par les opérations.

  Les données produites et consommées par les opérations se divisent en deux catégories, on appelle :

  – *donn´ees locales*, les données produites et consommées par la même opération,

  – *donn´ees communiqu´ees*, les données produites par une opération et consommées par une autre opération.

Règle 1 *Chaque sommet opérateur du graphe de l'architecture doit être connecté à un sommet mémoire RAM dédié au stockage des opérations du programme qu'il va exécuter séquentiellement. Cette mémoire peut être mixte ou non. Le contenu de ce sommet mémoire doit être préalablement chargé lors d'une phase d'initialisation appelée programmation.*

Règle 2 *Chaque sommet opérateur doit aussi être connecté à au moins un sommet mémoire RAM dédié au stockage de données. Les opérations exécutées par l'opérateur produiront ou consommeront des données dans ces mémoires connectées à l'opérateur. Cette mémoire peut être mixte (la même que celle du programme) ou non.*

1.2.2.4 Hiérarchie de mémoires RAM

Dans le paragraphe précédent nous avons encapsulé dans un même sommet les registres ayant des caractéristiques identiques. Si les registres ont des caractéristiques différentes (leurs bandes passantes sont différentes), ils sont encapsulés dans d'autres sommets mémoire RAM. Cela permet de distinguer, par exemple, les mémoires internes et les mémoires externes qui ont souvent des bandes passantes et des tailles différentes ainsi que les mémoires ROM [7], FLASH [8] etc. Pour connecter plusieurs mémoires RAM à un même opérateur il faut à nouveau multiplexer et démultiplexer les données par l'intermédiaire d'un Bus/Mux/Demux supplémentaire. Cela permet ainsi de modéliser (Cf. exemple de la figure 1.11) des hiérarchies de mémoires aux caractéristiques différentes (cela correspond à une hiérarchie de Bus/Mux/Demux puisque chaque mémoire RAM renferme aussi un Bus/Mux/Demux).

7. Read Only Memory
8. M´emoire effa¸cables ´electriquement

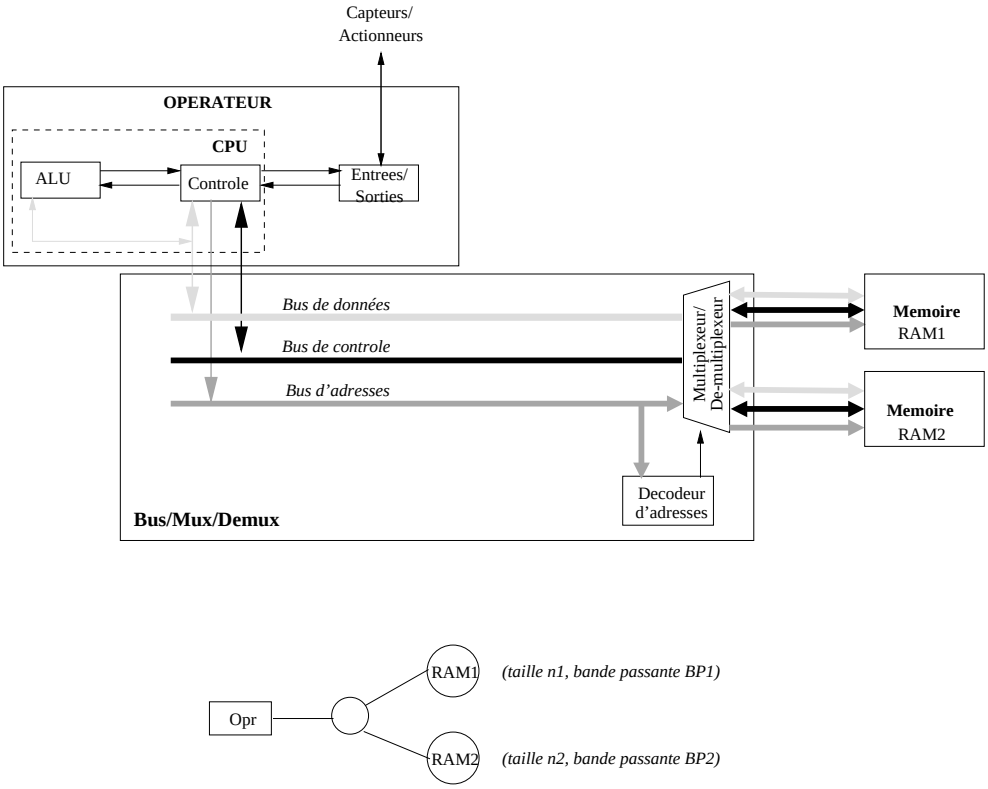


FIG. 1.11: *Hiérarchie mémoire*

1.2.2.5 Registre partagé

Dans les machines multi-opérateur, les opérateurs peuvent partager un registre pour s'échanger des données. Le partage de ce registre nécessite l'utilisation d'un Bus/Mux/Demux pour multiplexer/démultiplexer les données en provenance ou à destination d'un des opérateurs, mais aussi pour multiplexer les données en provenance des deux opérateurs. Pour empêcher les conflits d'accès en écriture ou en lecture par les opérateurs, le Bus/Mux/Demux est commandé par un *arbitre.* Cet arbitre gère les conflits d'accès mais n'impose pas d'ordre entre lecture et écriture qui reste donc aléatoire. L'arbitre étant indissociable du Bus/Mux/Demux qu'il contrôle, nous encapsulons l'ensemble Bus/Mux/Demux et arbitre dans un unique sommet Bus/Mux/Demux/Arbitre-RAM (Cf. figure 1.12). Ce sommet est caractérisé par sa politique d'arbitrage qui peut être à priorité fixe, tournante ou autre. Dans le cas des registres multiport, l'arbitre n'intervient que pour les conflits d'accès en écriture. Les données traversent alors une hiérarchie de Bus/Mux/Demux (le Bus/Mux/Demux/Arbitre-RAM et le Bus/Mux/Demux inclus dans le sommet mémoire RAM). La mémoire et le Bus/Mux/Demux/Arbitre-RAM qui permet de la partager peuvent être encapsulés à nouveau dans un sommet mémoire RAM qui possédera donc la caractéristique supplémentaire de son arbitre.

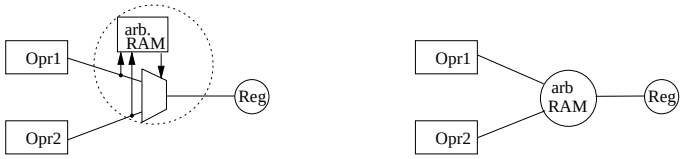


FIG. 1.12: *Registre partagé*

1.2.2.6 Mémoire RAM partagée

Plutôt que de partager un seul registre, les opérateurs peuvent partager un ensemble de registres encapsulés dans une mémoire RAM. Dans ce cas il faut aussi insérer un sommet Bus/Mux/Demux/Arbitre-RAM entre les opérateurs et la mémoire RAM alors qualifiée de partagée (Cf. figure 1.13). La politique d'arbitrage associée au sommet Bus/Mux/Demux/Arbitre-RAM définit l'accès aux registres de la mémoire qui reste aléatoire au niveau de l'ordre d'accès entre lecture et écriture. Nous pouvons encapsuler cet arbitre RAM et la mémoire RAM dans un unique sommet que nous appellerons *sommet m´emoire RAM partag´e.* La présence d'un arbitre RAM est donc implicite dans un sommet mémoire RAM partagé.

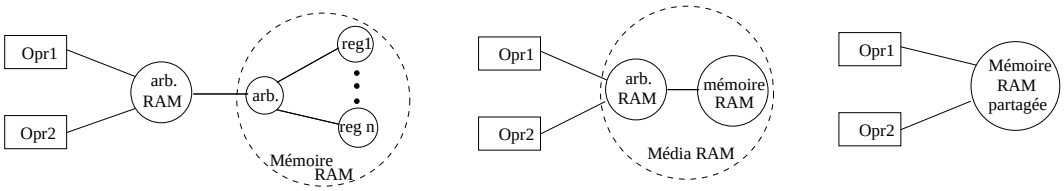


FIG. 1.13: *Mémoire RAM partagée*

Règle 3 *Un sommet mémoire RAM partagé est donc connecté au minimum à deux arcs dont l'extrémité peut être connectée soit à un sommet opérateur, soit à un sommet communicateur comme nous me verrons en 1.2.2.9.*

1.2.2.7 Registre partagé et arbitrage SAM

Dans le paragraphe 1.1.2.3 nous avons vu que les machines peuvent communiquer par passage de messages à l'aide de mémoires SAM. Rappelons que dans ces mémoires à accès séquentiel, les données y sont lues dans l'ordre de leur écriture, elles imposent que chaque opération d'écriture soit suivie d'une opération de lecture.

La plus simple de ces mémoires est composée d'un unique registre associé à un arbitre spécifique de type SAM (Cf. figure 1.14) pour le connecter à deux opérateurs. Comme précédemment c'est l'arbitre qui gère les conflits d'accès au registre, mais c'est aussi lui qui impose l'ordre entre écriture et lecture. L'arbitre particulier de type SAM et le Bus/Mux/Demux sont encapsulés dans un sommet Bus/Mux/Demux/Arbitre-SAM. Une fois qu'un opérateur a écrit une donnée dans le registre, l'arbitre SAM empêche tout accès en écriture par un opérateur, seul l'autre opérateur connecté à l'arbitre est autorisé à lire la donnée écrite.

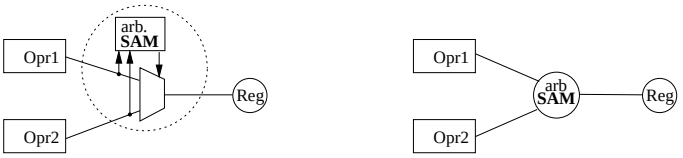


FIG. 1.14: *Registre partagé et arbitrage SAM*

L'arbitre peut être conçu de façon à rendre unidirectionnel l'accès au registre. Dans ce cas un seul opérateur, toujours le même, est capable d'écrire dans le registre, l'autre étant uniquement autorisé à y lire les données. Les arcs connectant les opérateurs à l'arbitre ne sont alors plus bidirectionnels mais orientés. Dans l'exemple de la figure suivante (figure 1.15), les données sont uniquement communicables dans le sens $Opr1$ vers $Opr2$.

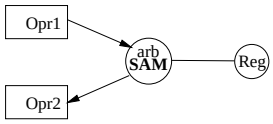


FIG. 1.15: *Arbitrage SAM monodirectionnel*

Remarque 5 *La mémoire SAM n'est jamais utilisée pour stocker les programmes (opérations) car un opérateur doit pouvoir y accéder aléatoirement (suite à un branchement par exemple), il faut donc toujours une mémoire RAM. Sa présence est implicite sur les deux figures précédentes car l'objectif de cette section est de mettre en évidence les communications inter-opérateurs.*

1.2.2.8 Mémoire SAM partagée

Une mémoire SAM à une place est construite par encapsulation du Bus/Mux/Demux/Arbitre-SAM et de l'unique registre (Cf. figure 1.16). La mémoire SAM obtenue peut être directement partagée par deux opérateurs. Cela correspond donc à une mémoire SAM point-à-point de taille 1 puisqu'elle est basée sur un seul registre.

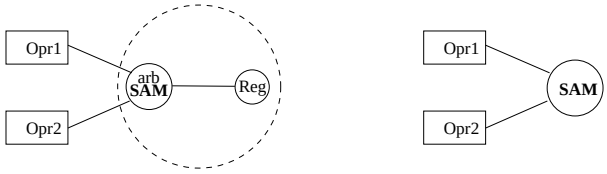


FIG. 1.16: *Mémoire SAM construite par encapsulation*

Les mémoires SAM peuvent être faites de plusieurs registres (cas des FIFO), cependant nous avons vu (Cf. § 1.1.2.3) que les données n'étaient pas distinguées spatialement par chaque opérateur qui accède aux données par une adresse unique. C'est l'arbitre qui contrôle l'accès aux différents registres en pilotant un Bus/Mux/Demux connecté à tous les registres. L'arbitre garanti que l'ordre de lecture des données est toujours identique à celui d'ordre écriture.

La figure suivante (1.17) présente les différentes étapes d'encapsulation qui mènent à un sommet mémoire SAM. Ainsi, le schéma $a)$ met en évidence le Bus/Mux/Demux/Arbitre-SAM qui est chargé d'assurer la précédence entre lecture et écriture de chaque donnée par les opérateurs. C'est aussi lui qui gère le stockage des données dans différents registres qui lui sont connectés par l'intermédiaire d'un Bus/Mux/Demux. Dans le schéma $b)$ nous avons encapsulé le Bus/Mux/Demux et les registres dans un sommet mémoire RAM comme expliqué dans le paragraphe 1.2.2.3. Dans le schéma $c)$ nous avons encapsulé le Bus/Mux/Demux/Arbitre-SAM et la mémoire SAM dans un unique sommet mémoire SAM.

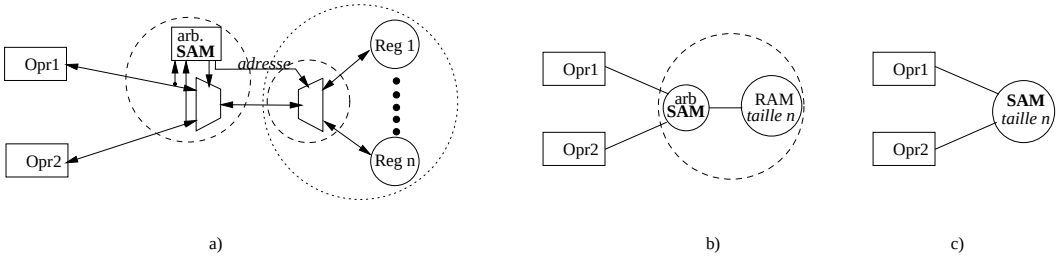


FIG. 1.17: *Mémoire SAM composée de plusieurs registres*

La mémoire SAM que nous venons de présenter est de type point-à-point, il existe aussi des mémoires SAM multipoint. Ces dernières peuvent être connectées à plusieurs opérateurs qui sont capables de lire ou d'écrire dans la SAM, l'arbitre garantissant toujours l'ordre d'accès entre lecture et écriture des données. Nous avons vu (Cf. § 1.1.2.3) qu'il existe deux types de mémoire SAM multipoint selon qu'elles supportent ou non la diffusion (broadcast) matériellement :

- quand une donnée a été écrite par un opérateur dans une mémoire SAM multipoint sans broadcast, elle ne peut être lue qu'une seule fois par l'un des autres opérateurs connecté à la mémoire,
- quand une donnée a été écrite par un opérateur dans une mémoire SAM multipoint avec support du broadcast, elle peut être lue simultanément par tous les opérateurs connectés à la mémoire.

Règle 4 *Un sommet mémoire SAM est donc connecté au minimum à deux arcs dont l'extrémité peut être connectée soit à un sommet opérateur, soit à un sommet communicateur comme nous allons le voir dans le paragraphe suivant.*

1.2.2.9 Communicateur

Intérêt Un communicateur représente un séquenceur autonome d'opérations de communication (transfert de données) entre mémoires RAM et/ou SAM. Les communicateurs sont donc toujours connectés à au moins deux mémoires, ils permettent d'améliorer les performances d'une architecture matérielle en libérant les opérateurs des tâches de communications, ils introduisent du parallélisme entre calcul (opérateur) et communication (communicateurs). En effet, prenons l'exemple d'une architecture composée de deux opérateurs connectés chacun à une RAM non partagée (locale), et communiquant au moyen d'une même mémoire RAM partagée (Cf. graphe *a* de la figure 1.18). Si la bande passante de cette mémoire partagée est plus faible que celle des mémoires locales, cela ralentit les opérateurs qui y accèdent pour se communiquer des données. L'ajout de communicateurs entre les mémoires locales et la mémoire partagée permet d'éviter un tel ralentissement (Cf. graphe *b* de la figure 1.18). Ainsi, lorsque l'opérateur "émetteur" transmet des données à l'opérateur "récepteur", il les écrit dans sa mémoire locale. C'est ensuite le communicateur qui transfère les données entre cette mémoire locale et la mémoire partagée. Enfin, ces données sont transférées dans la mémoire locale de l'opérateur récepteur par le second communicateur. Pendant que les communicateurs effectuent les transferts de données, les opérateurs peuvent consacrer leur temps à effectuer plus de calculs, il y a parallélisme(découplage) entre calcul et communication.

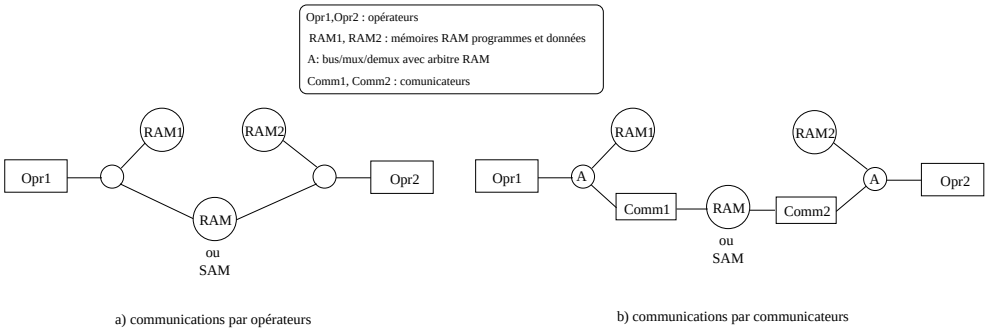


FIG. 1.18: *Architectures sans communicateur (a) et avec communicateurs (b)*

Les communicateurs présentent donc de l'intérêt lorsque les mémoires partagées par les opérateurs ont de plus faibles bandes passantes que celles des opérateurs. Dans le cas de mémoires RAM partagées, il faudra

donc bien observer les caractéristiques réelles de l'architecture lors de la modélisation. En revanche, quand la mémoire partagée est de type SAM, il est toujours préférable du point de vue des performances, de réaliser le transfert de données à l'aide de communicateurs. En effet, sans communicateur, la synchronisation entre lecture et écriture que requiert ce type de mémoire introduit un couplage fort entre les opérateurs émetteur et récepteurs qui sont obligés de se synchroniser. Les communicateurs, en libérant les opérateurs des tâches de communication permettent donc de les découpler. Les opérateurs peuvent donc passer plus de temps à calculer grâce au parallélisme entre calcul et communication introduit par les communicateurs.

Principes Un communicateur est pourvu d'un séquenceur autonome d'opérations de communication qui doivent être stockées dans une mémoire programme RAM connectée au communicateur (comme un opérateur). Une opération de communication est capable de transférer un registre mémoire, ou un ensemble de registres mémoire (i.e. une données ou un ensemble de données) entre deux mémoires. Pour cela elle génère des accès en lecture (génération d'adresses) pour l'une des mémoires et des accès en écriture pour la mémoire destinatrice afin d'y stocker les données lues. Quand le format des données utilisées par les opérateurs diffère (big indian, little indian), l'opération de communication doit aussi transformer les données de façon à les écrire dans le format de l'opérateur qui les utilisera. Comme les opérateurs, les communicateurs sont caractérisables (Cf. caractérisation § 4.1), par l'ensemble des opérations de communication qu'ils sont capables d'exécuter et par la durée d'exécution de chacune de ces opérations. Parmi ces opérations de communication, certaines servent uniquement à synchroniser les communicateurs et les copérateurs qui partagent des mémoires RAM. Nous verrons (Cf. § 6.2.2) que cela permet de synchroniser le début d'une opération de transfert de données avec la fin d'une opération de calcul productrice de données.

Règle 5 *Un communicateur est toujours connecté à au moins deux sommets qui peuvent être de RAM ou SAM partagé. Ce peut être par l'intermédiaire d'un ou plusieurs Bus/Mux/Demux ou Bus/Mux/Demux/Arbitre-RAM.*

Règle 6 *Les données ne peuvent être déplacées d'une mémoire à une autre que par un opérateur ou un communicateur. Des sommets RAM ou SAM ne peuvent donc être connectés directement entre eux.*

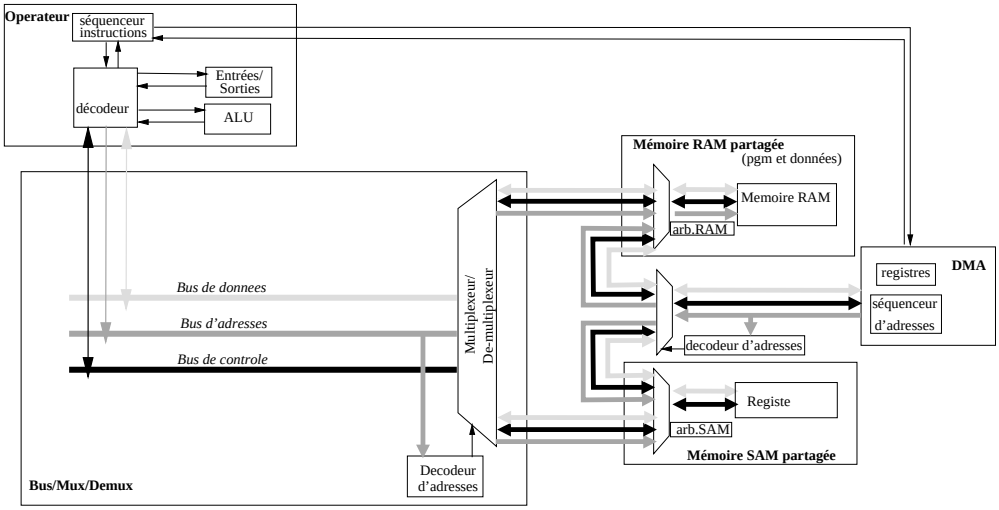


FIG. 1.19: *Exemple d'implantation d'un DMA*

Implantation par DMA A notre connaissance, il n'existe pas d'équivalent physique direct d'un communicateur tel que nous l'avons décrit, hormis dans le prototype de processeur qui a été développé sous forme de FPGA dans [95] ("SynDEx dans le silicium"). Dans les architectures actuelles comprenant un DMA, le communicateur modélise la coopération entre le séquenceur d'instructions d'un processeur et un canal DMA de ce processeur (Cf. figure 1.19). En effet, puisqu'un DMA requiert toujours le séquenceur pour programmer les registres de ses canaux, une petite partie de chaque opération de communication est exécutée par le séquenceur d'instructions pour programmer les registres (sous interruption de la séquence de calcul de l'opérateur qui programme les registres du DMA). Une fois programmé, le DMA peut prendre en charge seul le transfert de données correspondant à l'opération de communication. Dans les deux paragraphes suivant, nous allons étudier la modélisation d'une architecture avec DMA selon que les DMA partagent une ou plusieurs mémoires RAM avec l'opérateur :

- **DMA multicanaux partageant une unique RAM avec un opérateur :** généralement, chaque canal DMA est dédié à la gestion d'une mémoire (SAM ou RAM) utilisée pour les communications inter-opérateurs et a accès à une mémoire partagée par tous les canaux au travers d'un unique bus. Tous les canaux d'un DMA sont capables de transférer quasi-simultanément [9] des données entre la mémoire qui leur est dédiée et la mémoire partagée entre tous les canaux. Nous modélisons (Cf. 1.20) chaque canal DMA par un communicateur connecté d'une part à la mémoire à laquelle il est dédié, et d'autre part à un Bus/Mux/Demux/Arbitre-RAM connecté à la RAM partagé par tous les canaux.

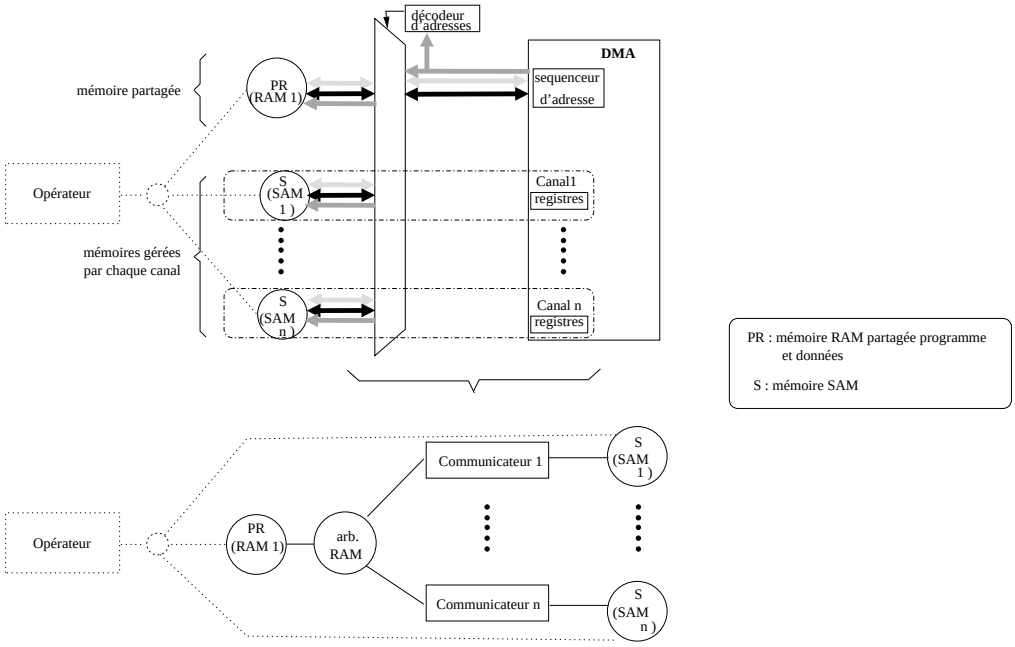


FIG. 1.20: *Modélisation d'une architecture avec DMA multicanaux*

- **DMA multicanaux se partageant plusieurs RAM :** si chacun des canaux du DMA a accès à plusieurs RAM (et non plus un unique comme précédemment), il faut ajouter un Bus/Mux/Demux entre le Bus/Mux/Demux/Arbitre-RAM et toutes ces mémoires comme il est indiqué sur la figure suivante

9. l'automate du DMA fait du time slicing en interla¸cant les transferts de fa¸con `a profi ter au mieux de la bande passante de chaque m´emoire

(1.21). Un DMA à plusieurs canaux est donc modélisé par un sous graphe constitué de communicateurs (autant que de canaux), d'un Bus/Mux/Demux et d'un Bus/Mux/Demux/Arbitre-RAM. Cela permet de mettre en évidence les séquences de transferts au niveau de chaque communicateur.

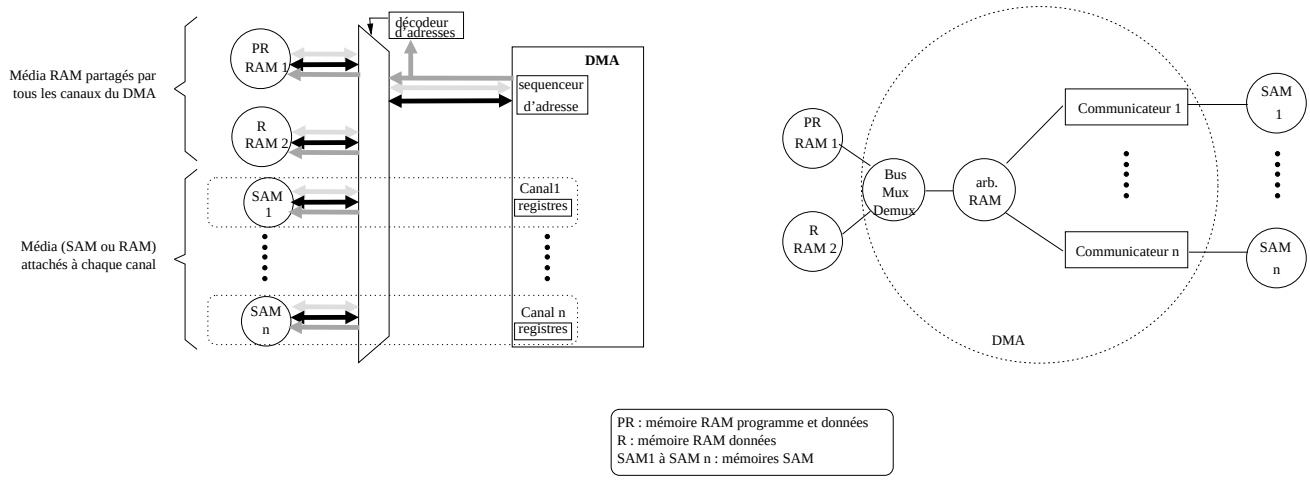


FIG. 1.21: *Modélisation d'une architecture avec DMA multicanaux connecté à 2 RAM*

Remarque 6 *Dans le premier cas il est possible d'encapsuler la RAM et le Bus/Mux/Demux/Arbitre-RAM dans un unique sommet RAM pour simplifier la spécification.*

Remarque 7 *Dans le second cas il est impossible d'encapsuler une RAM et le Bus/Mux/Demux/Arbitre-RAM dans un unique sommet RAM car le Bus/Mux/Demux/Arbitre-RAM est connecté à plusieurs RAM. Par contre il est possible d'encapsuler le Bus/Mux/Demux et le Bus/Mux/Demux/Arbitre-RAM dans un unique sommet Bus/Mux/Demux/Arbitre-RAM. Ce dernier modélise les bus et l'arbitre du DMA partagé par tous les canaux (communicateurs).*

Remarque 8 *Si le processeur ne possède pas de DMA, la séquence de communications du communicateur est entièrement réalisée par l'opérateur, au détriment des opérations de calcul qui seront plus souvent interrompues. Ce cas n'est donc intéressant que lorsque le volume des communications est faible vis à vis des volumes de calculs.*

1.2.2.10 Processeur

Comme cela a été expliqué dans la partie consacrée à l'état de l'art, il n'existe pas de définition stricte et unique d'un processeur. Dans notre modèle, un processeur est défini par un sous graphe constitué d'un opérateur et d'une mémoire RAM programme, ainsi que d'éventuels RAM données et communicateurs connectés à la RAM programme de l'opérateur. La séquence de calcul de l'opérateur et les séquences de communication des communicateurs sont stockées dans cette même mémoire RAM programme.

Processeurs basés sur une architectures Harvard

Les processeurs basés sur une architecture Harvard sont capables d'accéder simultanément à deux types de mémoires distinctes : la mémoire programme, qui sert uniquement à stocker le programme, et la mémoire de données. Pour cela il est équipé de deux bus physiquement indépendants. La figure suivante (figure 1.23) donne un exemple d'une telle architecture :

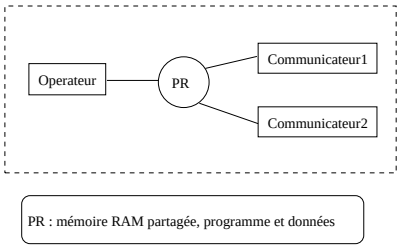


FIG. 1.22: *Exemple de processeur*

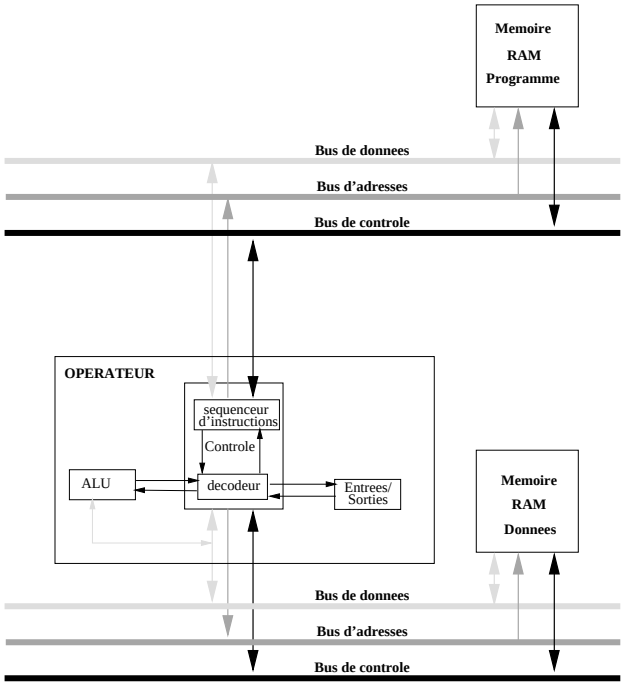


FIG. 1.23: *Architecture Harvard*

Pour mettre en évidence le parallélisme d'accès aux deux mémoires, nous connectons l'opérateur directement à deux mémoires RAM (qui peuvent être partagées avec d'autres opérateurs ou communicateurs) comme indiqué sur la figure 1.24 ci-dessous :

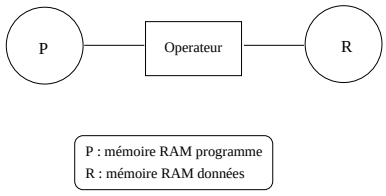


FIG. 1.24: *Modélisation d'architecture Harvard*

Processeur DSP

Les DSP sont souvent capables (ADSP21060, TMS320C40 . . .) de lire simultanément une instruction et d'accéder (en lecture ou écriture) à deux opérandes. Nous modélisons ces architectures par un sommet opérateur connecté à trois mémoires RAM dont deux sont dédiées aux données, la dernière étant dédiée au stockage du programme (figure 1.25) :

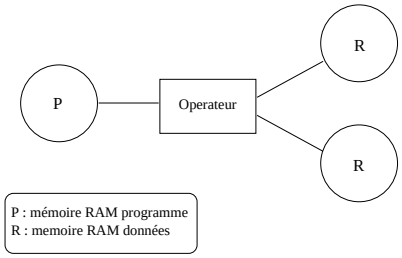


FIG. 1.25: *Modélisation d'un DSP*

1.2.3 Formalisation

Nous verrons dans les prochains chapitres que l'implantation d'un algorithme sur une architecture repose sur des heuristiques. Comme la complexité du problème d'implantation croit avec le nombre de sommets à manipuler, nous essaierons toujours, lors de la modélisation, d'encapsuler le plus de détails possibles afin de minimiser ce nombre de sommets. Les Bus/Mux/Demux/Arbitre-SAM seront donc toujours encapsulés, de même que les Bus/Mux/Demux/Arbitre seront uniquement nécessaires pour modéliser l'arbitrage des mémoires partagées par plusieurs communicateurs. Pour alléger le discours, à partir de maintenant nous utiliserons le mot SAM pour désigner une mémoire SAM, le terme RAM pour désigner les mémoires RAM partagées ou non et Bus/Mux/Demux/Arbitre pour les Bus/Mux/Demux avec arbitre. Un graphe d'architecture est donc décrit par G_{ar} un couple (S, A) où $S = S_{opr} \cup S_{com} \cup S_{RAM} \cup S_{bus} \cup S_{SAM}$, avec :

- S_{opr} l'ensemble des opérateurs,
- S_{RAM} l'ensemble des registres, mémoires RAM et mémoires RAM partagées, il se décompose en trois sous-ensembles : $S_{RAM} = S_{pgm}$ (mémoire programme) $\cup S_{data}$ (mémoire données) $\cup S_{mixte}$ (mémoire mixte),
- S_{bus} l'ensemble des Bus/Mux/Demux, Bus/Mux/Demux/Arbitre-RAM et Bus/Mux/Demux/Arbitre-SAM,
- S_{SAM} l'ensemble des mémoires SAM,
- S_{com} l'ensemble des communicateurs.

Soit μ_{opr} l'application qui, à chaque opérateur, associe l'ensemble des mémoire (RAM et/ou SAM) connectées :

$$\begin{array}{rccl} \mu_{opr} : & S_{opr} & \rightarrow & \mathcal{P}(S_{RAM} \cup S_{SAM}) \\ & p_i & \mapsto & \mu_{opr}(p_i) = m_j \end{array}$$

Soit μ_{opr}^{-1} l'application qui, à chaque RAM et SAM, associe l'ensemble des opérateurs connectés :

$$\begin{array}{rccl} \mu_{opr}^{-1} : & (S_{RAM} \cup S_{SAM}) & \rightarrow & \mathcal{P}(S_{opr}) \\ & m_i & \mapsto & \mu_{opr}^{-1}(m_i) = \{p_j \in S_{opr} / \mu_{opr}(p_j) = m_i\} \end{array}$$

Soit β_{opr} l'application qui, à chaque opérateur, associe l'ensemble des sommets Bus/Mux/Demux, avec ou sans arbitre, connectés :

$$\begin{array}{rccl} \beta_{opr}: & S_{opr} & \rightarrow & \mathcal{P}(S_{Bus}) \\ & p_i & \mapsto & \beta_{opr}(p_i) = b_j \end{array}$$

Soit β_{opr}^{-1} l'application qui, à chaque sommets Bus/Mux/Demux, avec ou sans arbitre, associe l'ensemble des opérateurs connectés :

$$\begin{array}{rccl} \beta_{opr}^{-1}: & S_{Bus} & \rightarrow & \mathcal{P}(S_{opr}) \\ & b_i & \mapsto & \beta_{opr}^{-1}(b_i) = \{p_j \in S_{opr} / \beta_{opr}^{-1}(p_j) = m_i\} \end{array}$$

Soit μ_{com} l'application qui, à chaque communicateur associe l'ensemble des RAM et/ou SAM connectées :

$$\begin{array}{rccl} \mu_{com}: & S_{com} & \rightarrow & \mathcal{P}(S_{RAM} \cup S_{SAM}) \\ & c_i & \mapsto & \mu_{com}(c_i) = m_j \end{array}$$

Soit ${\mu'_{com}}^{-1}$ l'application qui, à chaque RAM et SAM, associe l'ensemble des communicateurs connectés :

$$\begin{array}{rccl} \mu_{com}^{-1}: & (S_{RAM} \cup S_{SAM}) & \rightarrow & \mathcal{P}(S_{com}) \\ & m_i & \mapsto & \mu_{com}^{-1}(m_i) = \{b_j \in S_{com} / \mu_{com}(b_j) = m_i\} \end{array}$$

Soit β_{com} l'application qui, à chaque communicateur associe l'ensemble des sommets Bus/Mux/Demux, avec ou sans arbitre connectés :

$$\begin{array}{rccl} \beta_{com}: & S_{com} & \rightarrow & \mathcal{P}(S_{BUS}) \\ & c_i & \mapsto & \beta_{com}(c_i) = b_j \end{array}$$

Soit β_{com}^{-1} l'application qui, à chaque sommets Bus/Mux/Demux, avec ou sans arbitre associe l'ensemble des communicateurs connectés :

$$\begin{array}{rccl} \beta_{com}^{-1}: & S_{BUS} & \rightarrow & \mathcal{P}(S_{com}) \\ & b_i & \mapsto & \beta_{com}^{-1}(b_i) = \{c_j \in S_{com} / \beta_{com}(c_j) = b_i\} \end{array}$$

Soit μ_{bus} l'application qui, à chaque Bus/Mux/Demux et Bus/Mux/Demux/Arbitre associe l'ensemble des RAM et/ou SAM connectées :

$$\begin{array}{rccl} \mu_{bus}: & S_{bus} & \rightarrow & \mathcal{P}(S_{RAM} \cup S_{SAM}) \\ & b_i & \mapsto & \mu_{bus}(b_i) = m_j \end{array}$$

Soit $\mu_{bus}^{(-1)}$ l'application qui, à chaque RAM et SAM associe l'ensemble des Bus/Mux/Demux et Bus/Mux/Demux/Arbitre connectés :

$$\begin{array}{rccl} \mu_{bus}: & S_{RAM} \cup S_{SAM} & \rightarrow & \mathcal{P}(S_{bus}) \\ & m_i & \mapsto & \mu_{bus}^{-1}(m_i) = \{b_j \in S_{bus} / \mu_{bus}(b_j) = m_i\} \end{array}$$

Remarque 9 *Si pour un sommet $b_i \in S_{bus}$, $card(\beta_{opr}^{-1}(b_i)) = 1$ et $card(\beta_{com}^{-1}(b_i)) = \emptyset$ ou $card(\beta_{com}^{-1}(b_i)) = 1$ et $card(\beta_{opr}^{-1}(b_i)) = \emptyset$ le sommet ne possède pas d'arbitre car un seul sommet séquenceur est connecté.*

Remarque 10 *Si une mémoire m_i est telle que $(card(\mu_{opr}^{-1}(m_i)) + card(\mu_{com}^{-1}(m_i)) + card(\mu_{bus}^{-1}(m_i))) > 1$ cette mémoire est dite partagée.*

Un processeur p peut être défini par un opérateur $o_p \in S_{opr}$ et un ensemble de communicateurs noté $S_{com,p} \in S_{com}$.

1.2.4 Représentation graphique

Dans cette thèse nous avons fait le choix suivant pour représenter les graphes d'architecture :

- les sommets qui exécutent séquentiellement des opérations (i.e. les opérateurs et les communicateurs) sont symbolisés par des rectangles entourant leur nom : opr et com,
- un sommet Bus/Mux/Demux est symbolisé par un cercle vide ◯,
- un sommet Bus/Mux/Demux/Arbitre est symbolisé par un cercle vide en pointillé,
- une RAM, partagée ou non, dédiée uniquement au stockage d'un programme est symbolisée par la lettre "P" entourée d'un cercle : Ⓟ,
- une RAM, partagée ou non, dédiée uniquement au stockage et à la communication de données est symbolisée par la lettre "R" entourée d'un cercle : Ⓡ,
- une RAM, partagée ou non, dédiée au stockage de programmes et de données est symbolisée par les lettres "PR" entourée d'un cercle : ⓅⓇ,
- une SAM est symbolisée par la lettre "S" entourée d'un cercle : Ⓢ.

Graphiquement, si une RAM n'est connectée qu'à un seul arc, elle n'est pas partagée et n'inclue donc pas d'arbitre. Au contraire, dès lors qu'une RAM est connectée à plusieurs arcs (mémoire RAM partagée), l'arbitre est implicite.

1.3 Exemples de modélisations de machines

A partir des sommets présentés et des règles de connexions énoncées, il est possible de construire un grand nombre de machines, parallèles ou non.

1.3.1 Hiérarchie mémoire

La figure suivante (1.26) présente un opérateur capable d'accéder simultanément à deux mémoires de données parmi quatre, tout en lisant une instruction dans une mémoire programme séparée :

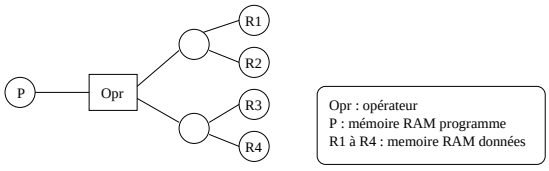


FIG. 1.26: *Architecture Harvard et hiérarchie mémoire*

1.3.2 Machines à mémoire RAM partagée

La machine, modélisée par le graphe ci-dessous (figure 1.27), est composée de trois opérateurs, pouvant chacun accéder à une RAM partagée parmi trois, au moyen d'un Bus/Mux/Demux/Arbitre. L'arbitre encapsulé dans chaque RAM peut par exemple être conçu de façon à privilégier un opérateur sur les trois qui y

sont connectés. Ainsi, R1 peut être assimilé à de la mémoire locale Opr1, R2 locale à Opr2 et R3 locale à R3 :

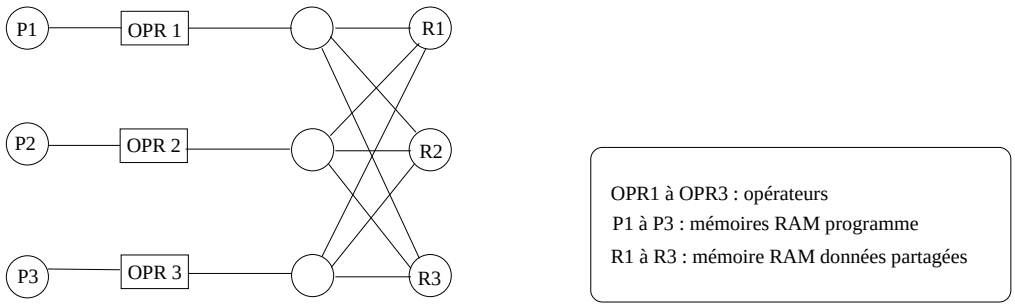


FIG. 1.27: *Machine à mémoire locale partagée*

1.3.3 Machine à mémoire locale et globale

L'architecture présentée sur la figure 1.28 est basée sur des opérateurs connectés à une RAM dite "privée" (puisque non partagée) et communicant par une unique RAM partagée :

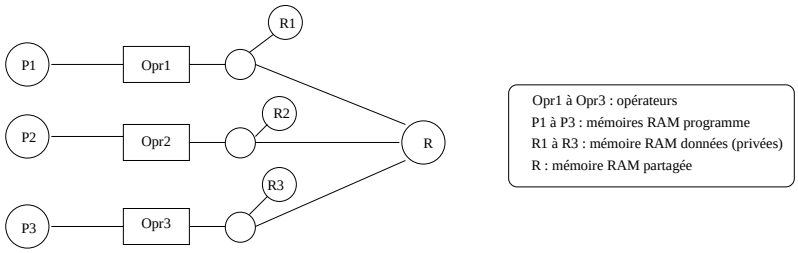


FIG. 1.28: *Machine à mémoires privées et partagées*

1.3.4 Machines à mémoire SAM

Dans l'architecture présentée sur la figure 1.29, chaque opérateur peut lire et/ou écrire, soit dans sa RAM privée, soit dans la SAM partagée. Il existe deux méthodes pour communiquer une donnée produite par une opération A exécutée par l'opérateur $Opr1$, utilisée par une opération B exécutée par l'opérateur $Opr2$:

1. soit l'opération A écrit directement les données qu'elle veut transmettre dans la SAM, et l'opération B lira ces données dans cette même mémoire,

2. soit l'opération A écrit les données à transmettre dans la RAM privée de opr1, puis ces données seront copiées dans la SAM par une opération spécifique exécutée par opr1. De là ces données seront copiées dans la RAM privée de opr2 par une opération spécifique. Enfin l'opération B pourra les lire dans la RAM privée.

La première méthode, semble plus simple et surtout plus efficace que la deuxième. Cependant elle oblige à concevoir les opérations A et B pour qu'elles écrivent leur résultat dans la SAM. Si le concepteur souhaite modifier l'implantation de façon à faire exécuter les opérations A et B par le même opérateur, il sera nécessaire de réécrire le code de ces opérations puisque pour se communiquer des données elles devront utiliser la mémoire RAM et non plus la SAM.

La deuxième méthode n'a pas cette contrainte. Quelque soit le placement des opérations, les données sont toujours stockées dans la RAM privée. Elles sont ensuite transférées, si nécessaire, par une opération générique de communication qui accède à la SAM et à la RAM. Cette deuxième méthode a cependant l'inconvénient d'ajouter potentiellement un surcoût mémoire puisque les données sont temporairement stockées dans la RAM même si elles sont communiquées par la SAM. Le temps de communication semble aussi allongé par cette méthode, mais nous verrons dans le chapitre consacré à l'optimisation que cela permet de découpler l'exécution des opérations de calculs des opérations de communications.

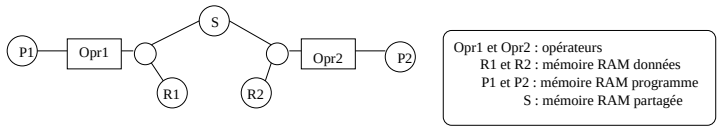


FIG. 1.29: *Machine distribuée basée sur le passage de messages*

Remarque 11 *Ces deux méthodes peuvent être aussi utilisées dans le cas où la RAM partagée est de type RAM et non de type SAM. Cependant, la deuxième solution n'a plus d'intérêt puisque le codage des opérations est identique dans les deux cas, seule l'adresse des données diffère.*

1.3.5 Communication point-à-point

Dans l'exemple de la figure 1.30, chaque opérateur est capable de communiquer par passage de message avec deux autres opérateurs au moyen de deux SAM point-à-point.

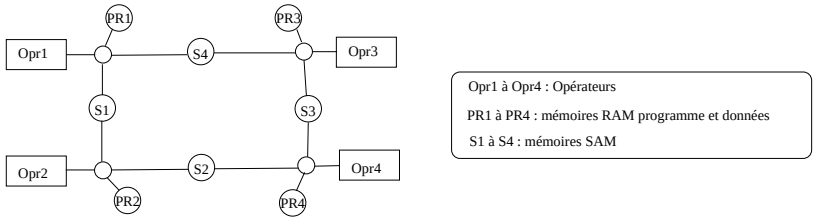


FIG. 1.30: *Machine distribuée basée sur les communications par passages de messages*

1.3.6 Communication par bus

La figure figure 1.31 présente un exemple classique de machine distribuée. Chaque opérateur dispose de sa propres mémoires, tous les opérateurs communiquent entre eux au moyen d'une unique SAM. Cette SAM peut par exemple modéliser un bus ethernet, un bus CAN, VAN, I2C etc..

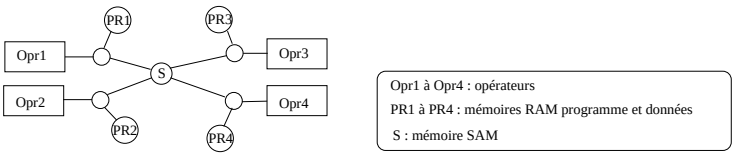


FIG. 1.31: *Machine distribuée à bus unique*

1.3.7 Architecture hétérogène

La machine modélisée dans la figure 1.32 repose sur trois opérateurs. Les opérateurs A et B communiquent par une SAM point-à -point, les opérateurs A, C et B, C communiquent par une RAM partagée.

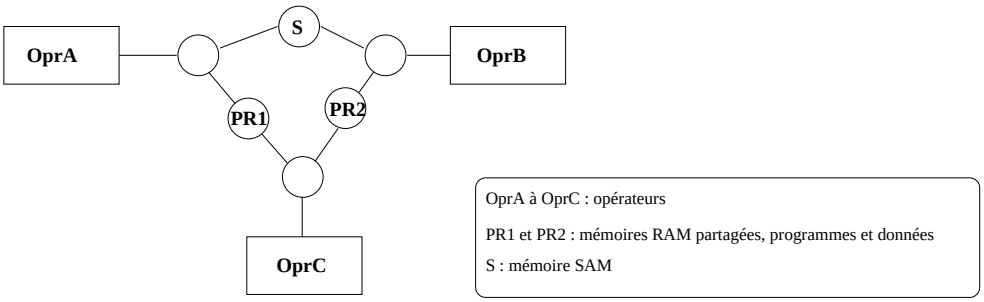


FIG. 1.32: *Communication par mémoire partagée et passage de messages*

1.3.8 Mémoire partagée et communicateurs

Dans le graphe de la figure 1.33, les deux opérateurs sont chacun connectés à une RAM partagée avec un communicateur. Les deux communicateurs sont connectés à une RAM qu'ils se partagent pour réaliser des transferts de données entre les opérateurs. Cette architecture modélise donc deux processeurs communiquant par une mémoire partagée. La mémoire partagée entre les communicateurs n'est pas accessible par les opérateurs, tous les transferts de données entre les opérateurs doivent passer par les communicateurs.

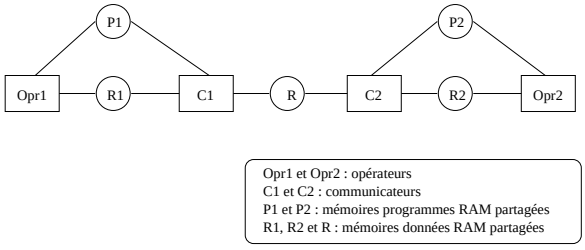


FIG. 1.33: *Machine distribuée avec communications par mémoire partagée et commmunicateurs*

1.3.9 Processeur de traitement du signal ADSP21060

L'ADSP 21060 produit par Analog Device, connu aussi sous le nom commercial "SHARC" pour "Super Harvard ARhitecture Computer" est le successeur de l'ADSP-21020. Le qualificatif "Super Harvard" lui vient du fait qu'il a les performances d'un processeur à 3 bus grâce à son cache mémoire instruction et la possibilité de mélanger programme et données dans ces bancs mémoire accessible simultanément au moyen de deux bus physiques. Le SHARC (Cf. bloc diagram figure 1.34) est divisé en quatre unités principales connectées ensembles par un crossbar :

- un CPU composé de trois unités de calcul parallèles sur des entiers et des flottants (ALU, MAC, Shifter), et de deux générateurs d'adresses indépendants (DAGs pour Data Address Generators)

- deux bancs de mémoire interne double port de 2Mbits chacun,
- une unité de communication incluant un contrôleur DMA reposant sur dix canaux, et connecté à six link ports pour des connections point-à-point rapide avec six autre ADSP21060 ou ADSP 21062, ainsi que deux ports série pour communiquer avec des périphériques ou d'autre SHARC,
- une port externe associé à une interface de communication.

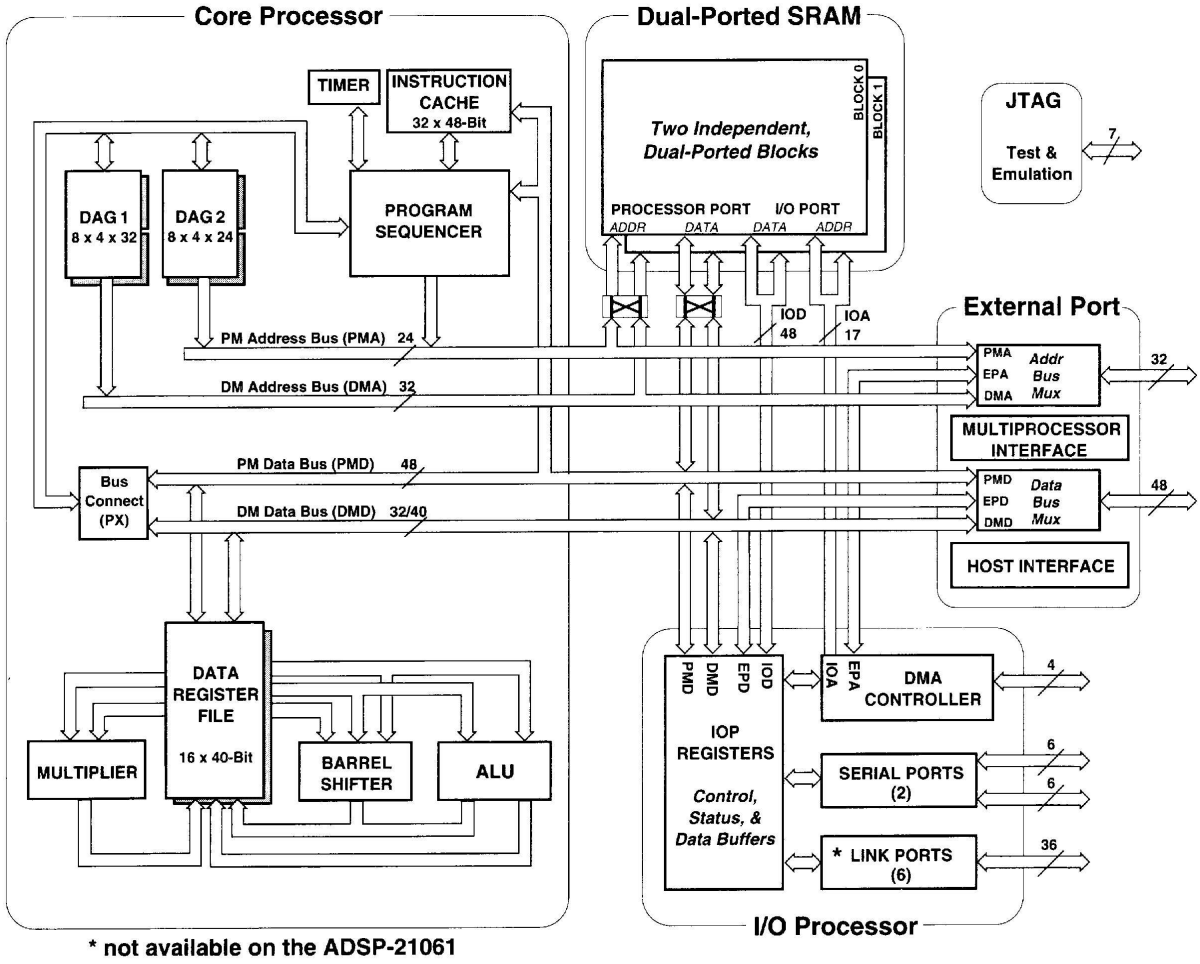


FIG. 1.34: *Block diagram d'un SHARC (ADSP21060)*

Ses principales caractéristiques sont :

- des mots d'instructions de 48 bits (qui peuvent contenir des mots immédiats de 32 bits ou le code de 3 instructions arithmétiques et un transfert mémoire) et gestion de données sur 16, 32 et 40 bits en format IEEE flottants et entiers sur 32 bits,
- une fréquence d'horloge de 40, 50 ou 60 MHz, lui donnant des performances crêtes de 120 (150, 180) MFLOPS.

La figure 1.35 présente le graphe correspondant à la modélisation de ce processeur. Le CPU, ses trois unités de calcul et ses deux générateurs d'adresses sont modélisés par un opérateur connecté à deux Bus/Mux/Demux ($b1$ et $b2$). Ces derniers permettent à l'opérateur d'accéder aux deux mémoires internes modélisées par deux sommets RAM ($PR1$ et $R2$) dont un seul permet le stockage de programme ($PR1$). De plus, $b1$ et $b2$ qui sont connectés à un Bus/Mux/Demux ($b4$) modélisent la possibilité qu'a l'opérateur d'accèder à chaque link port (sommets SAM $S1$ à $S6$) ainsi qu'aux deux ports série (sommets SAM $Ss1$ et $Ss2$) et à la mémoire externe (sommet RAM R). Le DMA et ses canaux sont modélisés par les communicateurs $Cserie1$, $Cserie2$, $Cextram$ et $C1$ à $C6$. Comme tous les communicateurs sont capable d'accéder aux deux mémoires RAM internes, il existe un arbitrage modélisé par le sommet Bus/Mux/Demux/Arbitre $b3$ connecté à tous les communicateurs et les deux RAM internes.

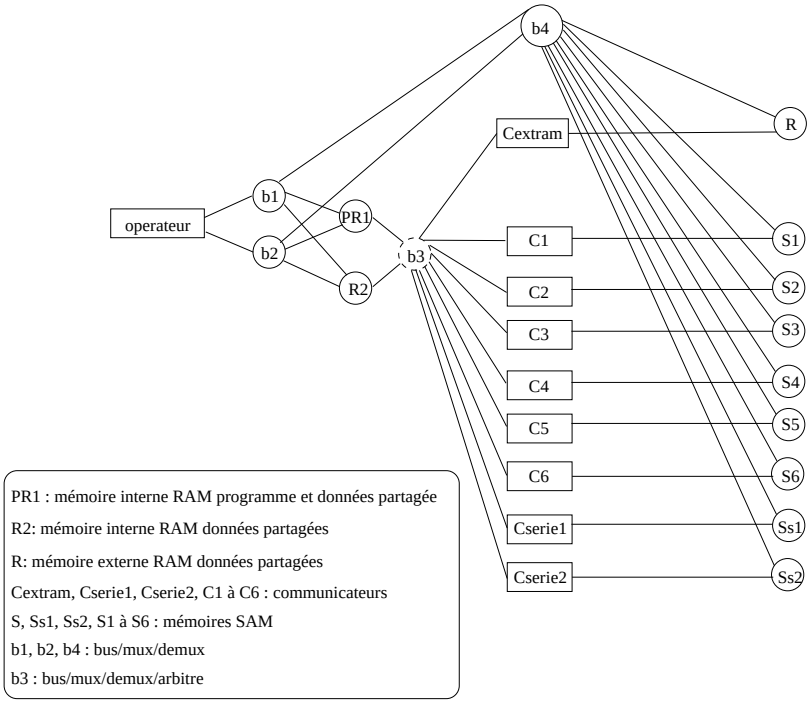


FIG. 1.35: *Modélisation d'un SHARC (ADSP21060)*

1.3.10 Multiprocesseur ADSP21060

La figure 1.36 présente une architecture basée sur quatre ADSP21060 communicant deux à deux par des SAM (liens point-à-point) ainsi que par RAM partagée. Pour ne pas surcharger le schéma, nous n'avons pas représenté les communicateurs et les SAM inutilisées pour les communications entre ces quatre processeurs.

Remarque 12 *Sur cette figure les opérateurs et communicateurs connectés à une même RAM programme (ils appartiennent donc au même processeur selon la définition donnée en 1.2.2.10) ont été englobés par des rectangles dessinés en pointillés. Ce peut être un moyen de spécifier visuellement les limites d'un processeur.*

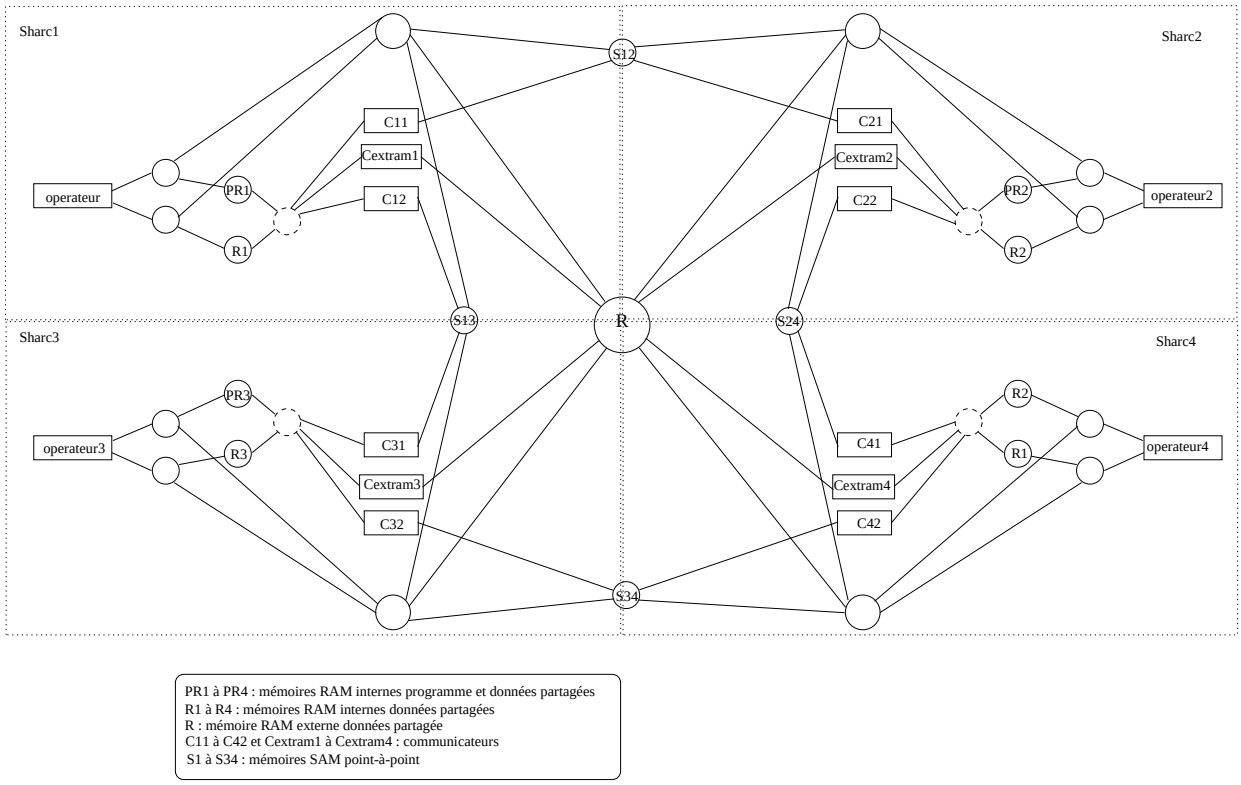


FIG. 1.36: *Modélisation d'une architecture composée de 4 Sharcs*

1.3.11 Processeur de traitement du signal TMS320C40

Le TMS320C40 est un DSP produit par Texas Instrument, son block diagram est présenté dans la figure 1.37. Il possède :

- deux bancs mémoire de données interne de 4 kilo-octets,
- deux ports d'accès à la mémoire externe (*global port* et *local port*) qui permettent d'adresser jusqu'à 16 Giga-octet de mémoire partagée programme et données,
- une mémoire cache interne programme de 512 octets,
- un CPU composé d'un controlleur mémoire et d'une unité de calcul en virgule flottante capable d'effectuer simultanément deux accès mémoires, une multiplication flottante, une opération arithmétique ou logique flottante, un branchement et la mise à jour d'un compteur de boucle,
- un DMA gérant 6 canaux dédiés à 6 ports de communications point-à-point pour des transferts de données à 20Mbytes/s, ou utilisable pour des transferts entre mémoires internes et/ou externes.

Toutes ces unités sont connectées par trois bus internes.

Comme le SHARC, il est adapté au fonctionnement multiprocesseur puisqu'il est capable de communiquer en mode point-à-point avec 6 autres C40 grâce à ses 6 ports de communications. Il est aussi capable de communiquer par mémoire partagée avec d'autres C40 grâce à ses deux ports mémoires externes. Pour

une fréquence de fonctionnement de 40MHz, il offre une puissance de calcul de 275 Méga-opérations par seconde (200MOPS par le CPU et 75MOPS par le DMA) et un débit de 320 Méga-octets par seconde (100Mo/s pour le port global, 100Mo/s pour le port local et 120 Mo/s pour les 6 ports de communications).

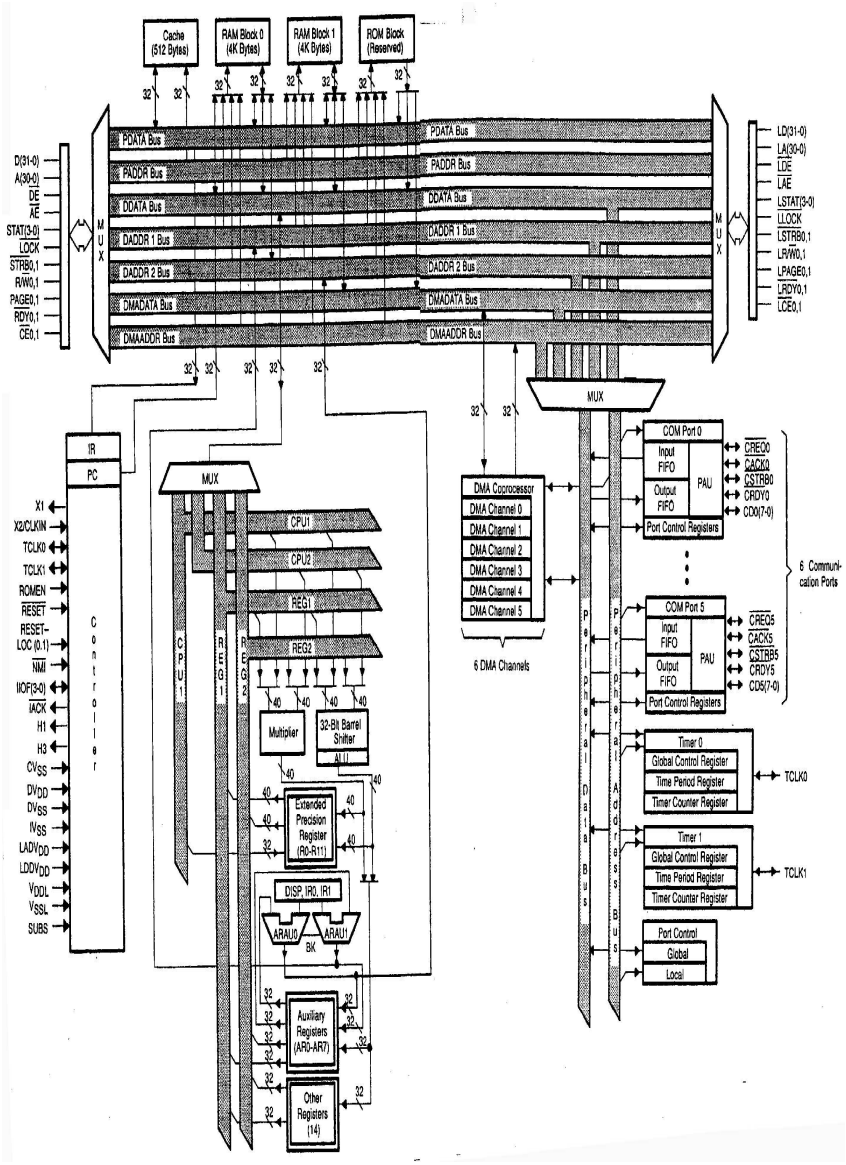


FIG. 1.37: *Block diagram d'un TMS320C40*

La figure 1.38 présente le graphe correspondant à la modélisation de ce processeur. Le CPU, son contrôleur mémoire et son unité de calcul sont modélisés par un opérateur. Comme il est capable d'accéder simultanément à deux mémoires internes et/ou externes modélisées par des sommets RAM ($R0$ et $R1$ pour les mémoires internes, R_{loc} et R_{glob} pour les mémoires externes éventuelles), l'opérateur est connecté à deux Bus/Mux/Demux ($b7$ et $b8$) qui sélectionnent les mémoires. Comme ces mémoires sont aussi accessible par chaque canal du DMA modélisés par des communicateurs ($C1$ à $C6$), chaque communicateur est connecté à ces mémoires internes par l'intermédiaire d'un Bus/Mux/Demux/Arbitre ($b9$) qui arbitre l'accès entre tous les communicateurs. Chaque port de communication point-à-point est modélisé par une SAM.

Chaque SAM étant à la fois accessible par un canal DMA ou par le CPU, elles sont connectées d'une part à un communicateur et d'autre part à l'opérateur. Le Bus/Mux/Demux $b10$ permet de sélectionner laquelle des SAM est accédée par l'opérateur. Les deux ports de mémoires externes permettent à l'opérateur et aux communicateurs d'accéder aux deux mémoires externes, il y a donc arbitrage que nous modélisons par deux sommets Bus/Mux/Demux/Arbitre ($b11$ et $b12$) entre les RAM externes, l'opérateur et les communicateurs. Les Bus/Mux/Demux $b1$ à $b6$ modélise la capacité, pour chaque communicateur, d'accéder soit à une SAM, soit à la mémoire externe.

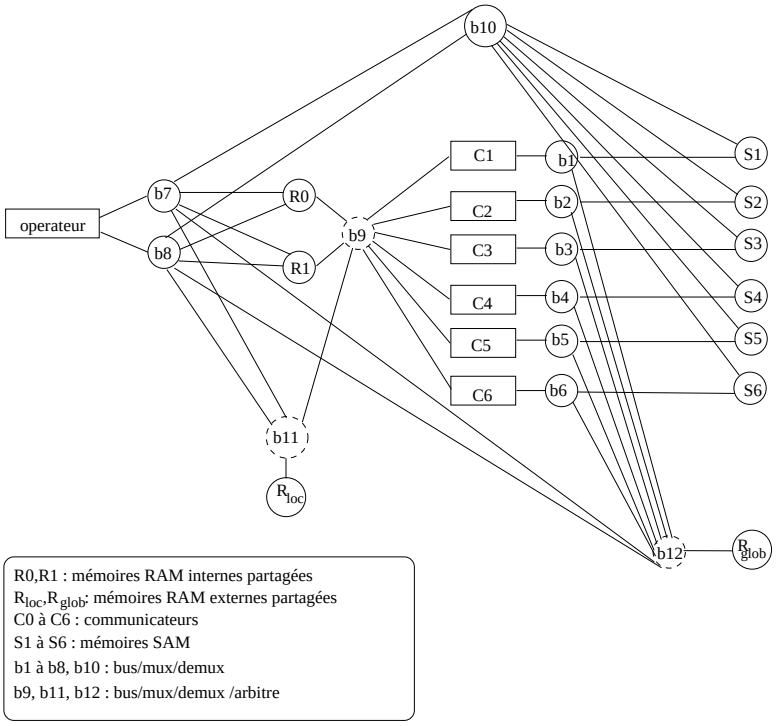


FIG. 1.38: *Modélisation complète d'un TMS320C40*

1.3.12 Multiprocesseur TMS320C40

La figure 1.39 présente un exemple d'architecture basé sur quatre TMS320C40 communicant deux à deux par des SAM (liens point-à-point) et tous ensemble par une unique mémoire RAM partagée R_{glob}. Dans cet exemple chaque C40 est connecté à une mémoire RAM externe non partagée (R_{loc}).

1.3.13 Processeur de traitement d'images TMS320C82

Le TMS320C82 est une version simplifiée du processeur TMS320C80, dédié aux applications multimédia, il renferme :

- un processeur Maître "MP" : processeurs RISC 32 bits avec unité arithmétique flottante (FP) IEEE-754,
- 2 processeurs parallèles [10] "PP" (PP0, PP1) : processeurs DSP 32 bits entier,

10. le TMS320C80 en poss`ede 4

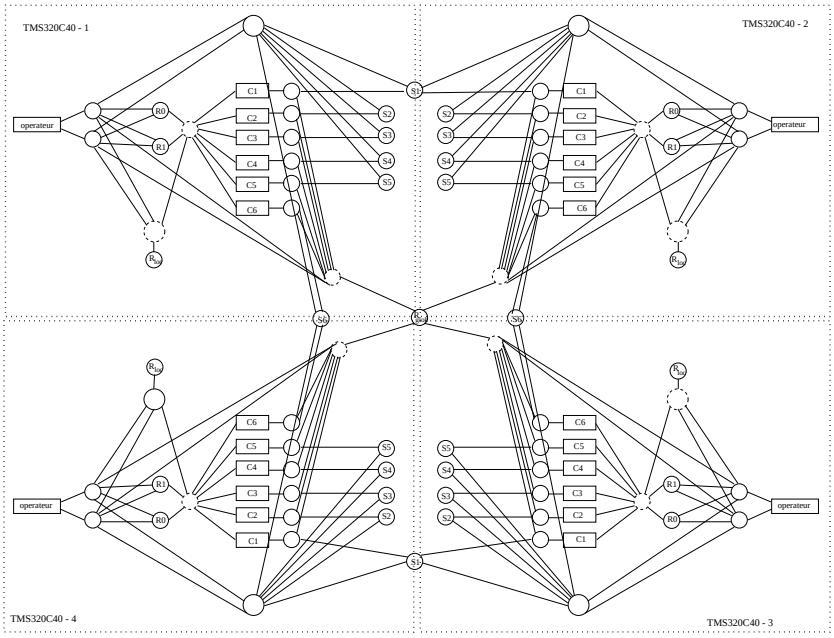


FIG. 1.39: *Modélisation d'une architecture composée de 4 TMS320C40*

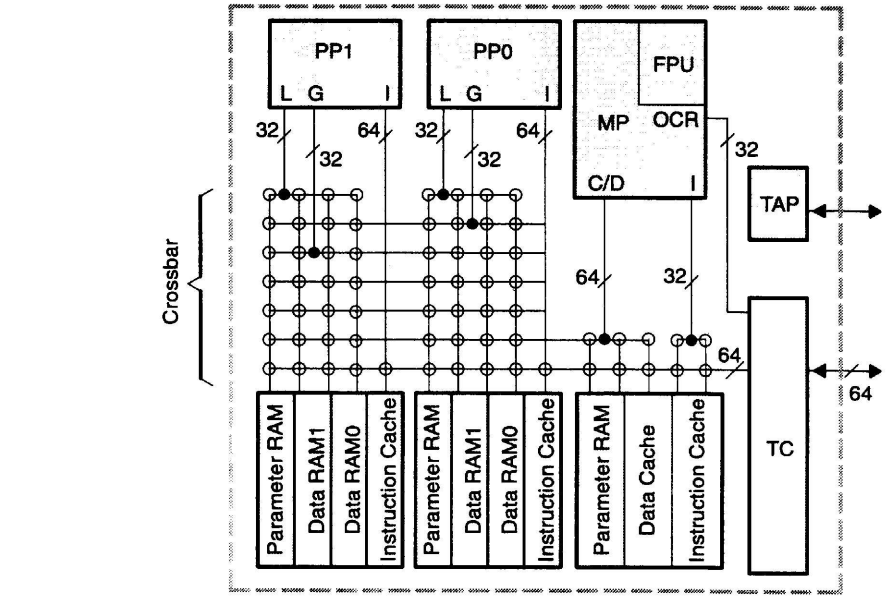


FIG. 1.40: *Block diagram d'un TMS320C82*

- 1 contrôleur de transfert (TC) : permet des transferts entre mémoires interne et externe en utilisant un bus 64 bits.[11]

ainsi que 11 bancs de mémoire "on-chip" divisés en plusieurs catégories :

- "Parameter RAM" : chaque processeur (MP et PP) possède un banc de 4KB de ce type de mémoire dédié à la mémorisation des vecteurs d'interruption mais qui peut aussi servir au stockage de données (Rpx),

- "Data RAM", chaque processeur parallèle (PP) possède 3 bancs de mémoire de 2Kb (Rdx-0 à Rdx-2) dédiés aux stockage des données. Chaque PP peut accéder à n'importe quel banc de mémoire au moyen du réseau d'interconnexion (crossbar),

- Cache instruction : chaque processeur (PP et MP) possède son propre cache instruction de 4KB (Px),

- Cache de donnée : le processeur MP possède 2 bancs de cache de données (Rd) programmables et contrôlés de façon logiciel.

La modélisation détaillée de ce processeur est donnée dans la figure 1.41 :

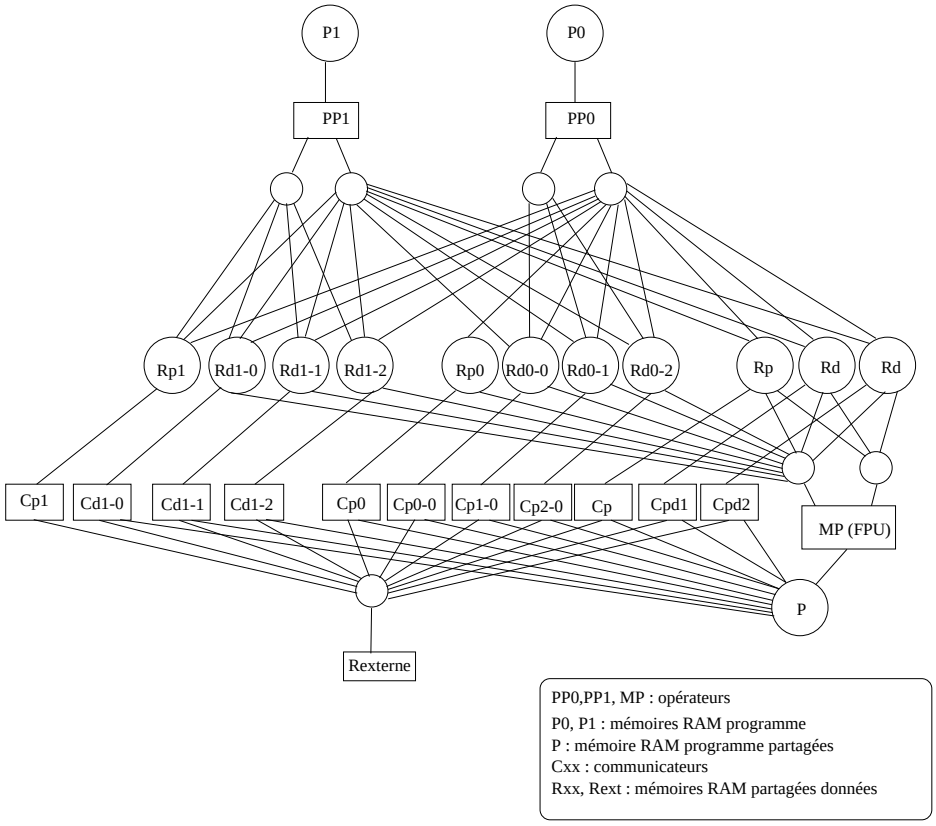


FIG. 1.41: *Modélisation d'un TMS320C82*

11. le TMS320C80 poss`ede en plus un contr^oleur vid´eo (VC) capable de faire de l'acquisition d'image

1.3.14 Architecture matérielle distribuée du Cycab

La figure 1.42 modélise l'architecture matérielle du Cycab, que nous présentons dans le paragraphe 11.2 (p. 217). Il est composé de cinq processeurs (MC68332) connectés par un bus CAN qui est une une mémoire SAM multipoint.

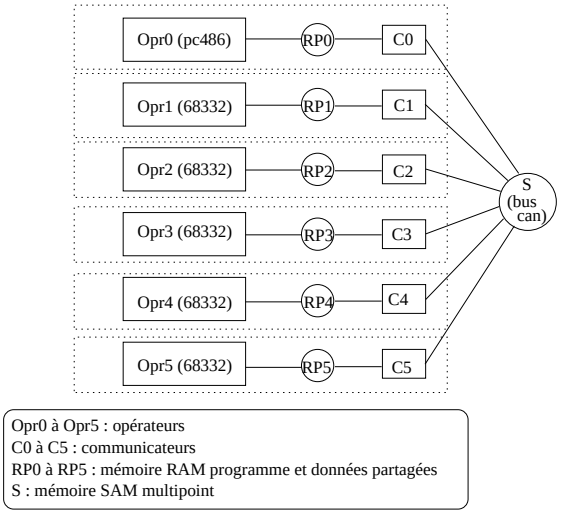


FIG. 1.42: *Modélisation de l'architecture matérielle du Cycab*

Chapitre 2

Modèle d'algorithme

Sommaire

Dans ce chapitre nous commencerons par présenter brièvement le modèle d'algorithme développé dans la thèse d'Annie Vicard[102], puis nous ajouterons quelques précisions sur la spécification du conditionnement, enfin, nous enrichirons ce modèle pour l'adapter à la génération automatique d'exécutif.

2.1 Hypergraphe Orienté

2.1.1 Modèle flot de contrôle et flot de données

De façon générale un algorithme peut être spécifié par un graphe ; on présente ci-dessous deux types de graphes : les graphes flot de contrôle et les graphes flot de données [41].

Dans un graphe flot de contrôle, les sommets du graphe sont des opérations qui consomment leurs données opérandes, et produisent leurs données résultats, dans des variables. Les arcs traduisent une relation d'ordre d'exécution "s'exécute avant" entre les opérations qu'ils relient. Dans la version "orientée automate", les sommets du graphe sont les états et les arcs définissent les transitions entre états, pendant lesquelles sont exécutées des opérations, qui elles aussi manipulent des variables. Dans les deux cas, un ordre total

d'exécution a été imposé sur l'ensemble des opérations du graphe, ce qui correspond bien à la définition standard donnée plus haut d'un algorithme. Pour pouvoir exécuter des opérations en parallèle, il faut faire une analyse de dépendance des données communiquées entre les opérations par l'intermédiaire des variables, afin de pouvoir décomposer l'algorithme en plusieurs graphes de contrôle (séquences d'opérations), composés en parallèle en établissant entre eux des communications au niveau des dépendances de données inter-séquences, comme dans le modèle CSP (Communicating Sequential Processes) de Hoare [51].

Nous modélisons l'algorithme par un graphe flots de données. Un graphe flots de données est un hypergraphe [39] orienté, où chaque sommet est une opération, et chaque arc est un transfert de données entre une opération productrice et une ou plusieurs (on parle alors de diffusion) opérations consommatrices. L'exécution de chaque opération et de chaque transfert de données est répétitive, d'où la notion de "flot". Chaque opération, à chacune de ses exécutions, consomme une donnée sur chacun de ses arcs d'entrée et les combine pour produire une donnée sur chacun de ses arcs de sortie. Une opération sans arc d'entrée (resp. de sortie) représente une interface d'entrée (capteur, resp. de sortie, actionneur) avec l'environnement physique. Lorsqu'une opération a besoin lors de sa n-ième exécution de consommer une donnée produite lors de la $(n-1)$-ième exécution d'une autre opération, il faut intercaler entre ces deux opérations une opération particulière appelée "retard" (le z^{-1} du traitement du signal), qui consomme une donnée sur son arc d'entrée après avoir produit sur son arc de sortie la donnée lue sur son arc d'entrée lors de son exécution précédente (une donnée initiale lors de sa première exécution). Sur chaque arc, chaque donnée doit être produite avant de pouvoir être consommée, donc les arcs traduisent une relation d'ordre d'exécution "s'exécute avant" entre les opérations, et en conséquence un graphe flot de données ne peut contenir de cycle que s'il y a au moins un retard dans chaque cycle. Ainsi un graphe flot de données n'impose qu'un ordre partiel sur l'exécution de ses opérations, et deux opérations qui ne sont pas en relation d'ordre peuvent être exécutées dans n'importe quel ordre, y compris en parallèle, si les ressources le permettent. De plus, la répétition implicite de l'exécution des opérations et des transferts de données autorise une autre forme de parallélisme "pipeline", où la n-ième donnée peut être produite sur un arc pendant que la $(n-1)$-ième est consommée, sur un autre processeur. Ces deux formes de *parall´elisme potentiel* permettent de nombreuses implantations d'un même algorithme, qui consistent chacune à composer différemment des opérations en séquence et les séquences en parallèle, avec communications inter-séquences comme dans le cas précédent.

L'intérêt principal d'un graphe flot de données est la description explicite des dépendances de données, nécessaires à l'implantation parallèle de l'algorithme. Il faut noter que pour les deux types de modèles, graphes flot de contrôle mis en parallèle, et graphes flot de données, on a étendu la notion initiale d'algorithme liée à un ordre total d'exécution sur les opérations, à un ordre partiel d'exécution. Par la suite nous utiliserons systématiquement ce modèle étendu quand nous parlerons d'algorithme.

2.1.2 Prise en compte du temps, vérifications, simulations

Le graphe de l'algorithme peut être obtenu par transformation de la spécification d'un algorithme exprimé dans un des langages synchrones : Esterel, Lustre, Signal, Statemate possèdent une sémantique tenant compte à la fois des aspects parallélisme et temporel [7, 69]. Alors qu'Esterel et Statemate sont des langages impératifs auxquels ont peut associer des graphes flot de contrôle, Lustre et Signal sont des langages déclaratifs auxquels ont peut associer des graphes flot de données.

Les langages Synchrones font l'hypothèse que les données produites par une opération apparaissent *simultan´ement* avec les données d'entrée qui ont déclenché l'opération, *c'est-à-dire sans attendre de nouvelles donn´ees d'entr´ee*. Cette notion de simultanéité est purement logique, elle se veut indépendante de toute implantation et donc des durées physiques d'exécution des opérations, durées qui dépendent de l'implantation. Cette hypothèse permet de considérer que les calculs (sommets du graphe flot de données) et les transferts de données (arcs du graphe flot de données) ont lieu de manière instantanée, leurs durées physiques ne sont

pas considérées. Par transitivité appliquée à tous les sommets du graphe, les données produites par le graphe apparaissent simultanément avec les données y entrant. Ces dernières venant de l'environnement définissent des événements d'entrée ou stimuli. De même, les données produites pour l'environnement par le graphe définissent des événements de sortie ou réactions. Tout événement de sortie est associé à un événement d'entrée. Cela permet de définir un temps logique où seul compte l'ordre relatif des événements, indépendamment des durées physiques qui s'écoulent entre les événements. La notion de durée (logique) n'existe alors qu'au travers du comptage des événements.

Les compilateurs des langages Synchrones effectuent des vérifications sur la cohérence entre les événements produits en réaction aux événements qui les déclenchent. À ce niveau les vérifications ne portent que sur l'ordre des événements, la notion de durée physique liée à une horloge temps réel n'est pas prise en compte. Cela sera cependant fait plus tard lors de l'implantation et sera décrit au chapitre 4. Les compilateurs permettent de montrer par exemple que certains événements auront toujours lieu, ou bien se produiront après un certain nombre d'occurrences d'un autre événement, ou bien que certains événements n'auront jamais lieu. Les raisonnements formels utilisés ici ne portent que sur des booléens, ils sont principalement basés sur des techniques de "Model Checking" utilisant le plus souvent des BDD (Binary Decision Diagram) [14]. Ces vérifications bien que limitées, éliminent un grand nombre d'erreurs logiques qui habituellement sont découvertes lors des tests en temps réel sur le prototype. Découvrir ces erreurs le plus tôt possible dans le processus de conception des applications temps réel embarquées est très important ; cela permet de diminuer la phase de tests temps réel très coûteuse que l'on estime parfois à 70% du temps de développement de ce type d'applications. On verra plus loin comment conserver ces propriétés lorsqu'on prendra en compte le temps physique au moment de l'implantation.

En plus des vérifications vues ci-dessus les compilateurs des langages Synchrones sont capables de générer un code exécutable séquentiel, généralement du C mais aussi du Fortran ou de l'Ada ou d'autres langages séquentiels. Ce code séquentiel est utilisé pour faire la simulation numérique et la vérification du comportement événementiel, en termes d'ordre sur les événements seulement, de l'algorithme ainsi spécifié.

2.1.2.1 Hypergraphe

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-opérations. Chaque opération est une suite indivisible d'instructions, appelée région atomique dans [108], 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.

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-opérations*. 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 (appelée aussi productrice) $\gamma^{-1}(d'_i)$ et on définit la fonction γ qui, à chaque

dépendance d'_i, associe l'ensemble des opérations réceptrices (appelées aussi consommatrices).

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

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

Le graphe est sans circuit. Il n'existe pas de chemin ayant à la fois même origine et même destination. Chaque dépendance de données a un et un seul émetteur et au moins un récepteur, ainsi :

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

Remarque 13 *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, *d'_i diffuse ses données.*

2.1.2.2 Relations entre opérations

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

- *Successeurs d'une opération o'_i au sens Gondran & Minoux [39] :* $\Gamma(o'_i)$
  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*.

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

- *Descendants d'une opération o'_i au sens Gondran & Minoux [39] :* $\hat{\Gamma}(o'_i)$
  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.

- *Prédécesseurs d'une opération o'_i au sens Gondran & Minoux [39] :* $\Gamma^{-1}(o'_i)$
  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*.

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

- *Ancêtres d'une opération o'_i au sens Gondran & Minoux [39] :* $\hat{\Gamma}^{-1}(o'_i)$

  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}. Tout d'abord nous définissons les opérations dites *logiquement d´ependantes* [19] et les opérations dites *logiquement ind´ependantes* [19] ou encore *concurrentes* [108]. Parmi les opérations dites *logiquement d´ependantes*, on distingue les opérations dites données-dépendantes et les opérations dites *transitives-dépendantes*. L'ensemble des opérations données-dépendantes, noté $\preceq$ est défini par :

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

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. Elle définit une relation d'ordre partiel d'exécution sur l'ensemble des opérations O'. Soit $\preceq^*$, la fermeture transitive de la relation $\preceq$. Cette relation définit un ordre partiel d'exécution sur l'ensemble des opérations du graphe de l'algorithme. L'ensemble $\preceq_T$ des opérations transitives-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. Finalement, l'ensemble des opérations logiquement dépendantes est caractérisé par la fermeture transitive $\preceq^*$ de la relation $\preceq$.

L'ensemble des opérations logiquement indépendantes est l'ensemble, noté $\geqslant$, des opérations O' appartenant au complémentaire de la relation $\preceq^*$. Cet ensemble $\geqslant$ caractérise le parallélisme potentiel de l'algorithme et on cherchera à l'exploiter si le parallélisme disponible de l'architecture le permet.

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

$$\xi \supseteq \preceq^*$$

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

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

2.1.3 Choix de la granularité

Dans les prochains chapitres, nous verrons comment exploiter le parallélisme potentiel d'un algorithme et le parallélisme disponible de l'architecture afin de réaliser une implantation efficace de l'application. Ce problème d'optimisation correspond à un problème d'allocation de ressources dont l'espace des solutions à explorer varie exponentiellement avec le nombre de sommets de l'architecture et de l'algorithme. Pour résoudre ce problème en temps raisonnable et compatible avec le prototypage rapide, nous utiliserons des heuristiques (Cf. chapitre 4). Néanmoins, comme la durée d'exécution de ces heuristiques est directement

liée au nombre de sommets qui composent les graphes d'algorithme et d'architecture, il est nécessaire de minimiser le nombre total de sommets des deux graphes. Pour cela nous utilisons l'encapsulation, comme pour la modélisation de l'architecture. Chaque sommet du graphe d'algorithme correspond à un grain indivisible de distribution et d'ordonnancement. Chaque grain est composé d'un ensemble d'instructions préordonnancées (correspondant par exemple à des séquences d'instructions issues de la compilation séparée de sous-programmes FORTRAN ou de fonctions C) que l'on a choisi d'appeler *op´eration* et que l'on peut aussi qualifier de *macro-instructions*. Les *d´ependances de donn´ees* correspondent à des agrégats de cellules mémoire contiguës (correspondant par exemple à des tableaux ou à des structures C) que l'on peut aussi qualifier de *macro-registres*. La taille d'un grain est proportionnelle au nombre d'instructions qu'il renferme, mais aussi de la quantité de données qu'il manipule. Cette approche macroscopique offre les avantages de :

- **portabilité :** les opérations qui composent l'algorithme doivent être portables entre processeurs ayant des jeux d'instructions différents, non seulement parce que la durée de vie d'un algorithme dépasse presque toujours celle de sa première implantation, mais aussi parce que l'architecture peut être hétérogène,
- **modularité :** les algorithmiciens préfèrent concevoir leurs algorithmes en termes de structures de données (vecteurs, matrices, listes ...) et d'opérations (filtres, transformées, opérations matricielles ...) plus complexes que celles directement supportées par les jeux d'instructions,
- **surcoût par grain :** la portabilité et la modularité impliquent un surcoût d'interface pour chaque opération (lecture des opérandes et écriture des résultats en mémoire, et construction du contexte d'appel pour les opérations compilées séparément) ; de même, chaque communication implique un surcoût d'initialisation du contexte de la communication ; plus les grains sont petits, plus ils sont nombreux, et plus le surcoût total augmente,
- **caractérisation :** de plus en plus de processeurs utilisent des mémoires caches et des pipelines pour chercher, décoder et exécuter leurs instructions, et/ou pour lire leurs opérandes et stocker leurs résultats, ce qui induit des interactions, entre instructions successives, qui dépendent de détails architecturaux complexes et souvent non publiés ; la variation relative de durée d'exécution d'une séquence d'instructions est moindre que celle d'une instruction isolée, car ces interactions ont plus tendance à se compenser qu'à se cumuler.

Cependant, cette agrégation ne doit être ni excessive ni déséquilibrée : si quelques gros grains représentent à eux seuls la quasi-totalité du volume de calcul et/ou s'il y a trop peu de grains par rapport au nombre de séquenceurs d'instructions, il y a peu de choix pour allouer les ressources et trouver une implantation efficace ; réciproquement, avec des grains plus petits, donc plus nombreux, le choix est beaucoup plus grand, parfois trop, la solution est alors très longue à trouver, on risque de cumuler plus d'erreurs, et le surcoût total augmente (car il y a un surcoût et une incertitude de durée d'exécution par grain, peu dépendants de la taille du grain, Cf. § 4.1). La pratique montre que le choix de la granularité a un impact important dans tous les problèmes d'optimisation d'allocation de ressources. La méthodologie "Adéquation Algorithme-Architecture" présentée dans cette thèse doit apporter une aide au choix de la granularité.

2.2 Factorisation

Certains algorithmes impliquent des volumes de calculs suffisamment importants pour générer des répétitions périodiques de traitements identiques (sur des données différentes), que l'esprit humain, se lassant vite des énumérations, préfère spécifier sous forme de graphes factorisés. Ce type de spécification, ne se restreint pas uniquement à réduire la taille de la spécification algorithme, il permet également de décrire

en intention [28] plusieurs implantations plus ou moins séquentielles ou parallèles, chacune avec des caractéristiques différentes. Ainsi, dans le cas d'une implantation distribuée d'un graphe factorisé, il est par exemple possible de tirer partie du recouvrement entre les calculs et les communications pour diminuer la durée totale d'exécution d'une implantation. La possibilité de spécifier explicitement les parties répétitives d'un graphe d'algorithme à l'aide de sommets de factorisation a donc été introduite [65] dans le modèle d'algorithme.

2.2.1 Factorisation finie

Afin de mieux comprendre cette notion de factorisation, de graphes factorisés, nous allons vous présenter un exemple simple d'algorithme faisant apparaître des parties régulières pouvant être factorisées à savoir le produit d'une matrice par un vecteur. En effet, le produit d'une matrice 3x3 *c* par un vecteur *e* de dimension 3 peut se décomposer en 3 produits scalaires qui peuvent se décomposer chacun en une somme de 3 produits. Ainsi, en terme de graphes, nous aboutissons aux représentations suivantes :

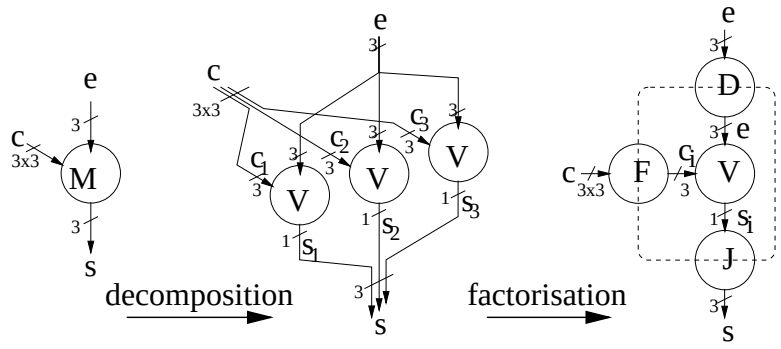


FIG. 2.1: *Décomposition et factorisation d'un produit matrice-vecteur*

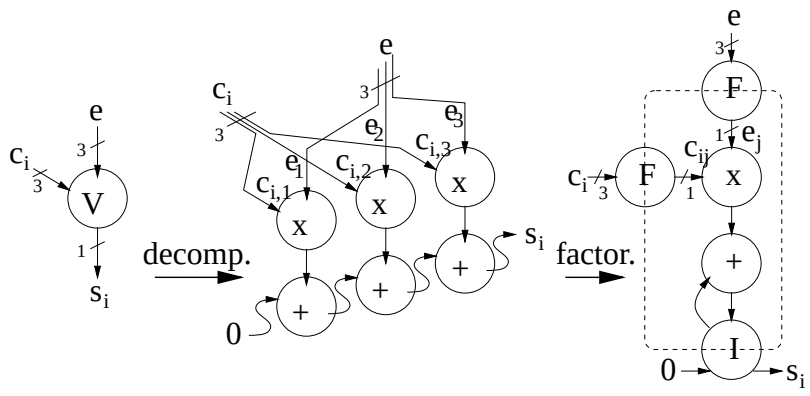


FIG. 2.2: *Décomposition et factorisation d'un produit scalaire*

La factorisation du produit matrice-vecteur (figure 2.1) fait apparaître trois sommets spéciaux (D pour "Diffusion", F pour "Fork" et J pour "Join") qui délimitent la frontière. Cette dernière est mise en évidence par des pointillés, entourant le motif de la factorisation pour le séparer du reste du graphe. Quant à la factorisation du produit scalaire (figure 2.2), elle fait apparaître le nouveau sommet frontière I (pour "Iterate"). Soit

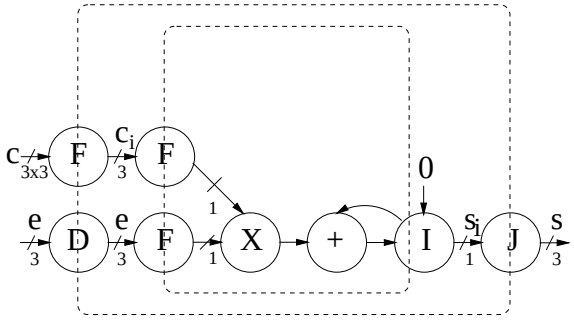


FIG. 2.3: *Graphe factorisé complet du produit matrice-vecteur*

$O'_{Diff}, O'_{Fork}, O'_{Join}, O'_{Iter}$ ces sous-ensembles de O'. Chaque sommet frontière spécifie ainsi une manière différente de traverser la frontière de factorisation :

- le sommet D "entre" en diffusant le vecteur *e* à tous les produits scalaires,
- le sommet F "entre" en factorisant le groupe d'arcs d'entrée,
- le sommet J "sort" en factorisant le groupe d'arcs de sortie,
- le sommet I "entre et sort" (par 0 et s_i) en factorisant le groupe d'arcs inter-motifs.

La factorisation du graphe ne change en rien sa sémantique opératoire, elle ne fait que réduire la taille de la spécification du graphe et mettre en évidence ses parties régulières (motifs périodiques) dont la connaissance ouvre la voie aux possibilités d'optimisation. Les sommets frontières ne sont que des délimiteurs d'une "syntaxe graphique", analogue aux parenthèses qui délimitent la portée d'un foncteur (opération sur les opérations) dans la syntaxe algébrique. De la même manière que pour deux paires de parenthèses, deux frontières de factorisation ne peuvent être que soit l'une dans l'autre, soit l'une à coté de l'autre, elles ne peuvent être en relation d'intersection. Une opération à l'intérieur d'une frontière de factorisation, qui représente un groupe factorisé d'opérations identiques opérant sur un groupe factorisé de données différentes, se réalise directement avec un seul opérateur utilisé itérativement, autant de fois qu'il y a d'opérations dans le groupe factorisé. Chaque groupe factorisé de données doit donc être multiplexé à la frontière : alors qu'ils ne jouent qu'un rôle "syntaxique" au niveau du graphe, les opérateurs correspondant aux sommets frontières doivent réaliser le multiplexage, sauf l'opérateur D qui se contente de fournir la même valeur à chaque itération.

2.2.2 Factorisation infinie et sommet retard

Nous avons vu que les systèmes réactifs interagissent avec leur environnement de manière discrète, donc répétitive (répétition de la séquence acquisition-calculs-commande). Cette répétition présente pour particularités notamment de contraindre les données d'entrée et les résultats de sortie à être multiplexés (le plus souvent périodiquement) à travers les capteurs et actionneurs d'interface avec l'environnement. Les sommets frontière F, J et I peuvent être étendus à la dimension infinie, il doivent alors avoir une autre implantation lorsqu'ils assurent l'interface avec l'environnement.

L'exemple de la figure 2.4 présente un graphe flot de données factorisé explicitement. L'opération *C* représente l'algorithme de contrôle commande (non décomposé) qui à chaque réaction *t* acquiert une nouvelle

entrée e_t et par l'intermédiaire du capteur E, la combine avec l'entrée z_t dont la valeur est égale à la sortie i_{t-1} de la réaction précédente (égale à la valeur *ini* lors de la réaction initiale), pour fournir la sortie s_t à l'actionneur S et la sortie i_t à l'entrée z_{t+1} de la réaction suivante.

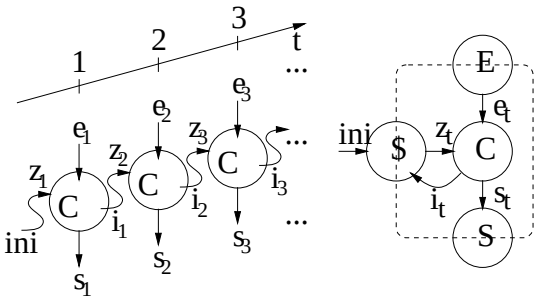


FIG. 2.4: *Graphe flot de données infiniment factorisé*

Nous considérons que la spécification d'un algorithme est toujours implicitement factorisée, les sommets *E* (capteurs) et *S* (actionneurs) réalisent materiellement de "fork" et le "join" des données. Nous appelerons sommet *retard* le sommet *iterate* "$" qui permet d'implanter les dépendances de données inter-itération du graphe d'algorithme infiniment factorisé. Soit $O'_{\$}$ le sous-ensemble de O' modélisant ce type d'opération.

2.3 Sommet Constante

Pour pouvoir paramétrer un algorithme, il peut être nécessaire de spécifier explicitement des données constantes en entrées de certaines opérations. Ces constantes requierent parfois d'être calculées, elles sont donc modélisées par un sommet opération. On remarque cependant que comme à chaque itération elles doivent fournir les même données aux opérations consommatrices, il n'est pas nécessaire de les ré-exécuter. Il suffit de conserver les valeurs calculées. C'est pourquoi nous avons ajouté un type particulier de sommet opération - les sommets *constante* - qui présente la particularité de n'avoir à être exécuté qu'une seule fois, même si ils font partie d'un graphe factorisé puisqu'à chaque exécution ils produisent les même données. Soit O'_{const} ce sous-ensemble d'opérations de O'.

2.4 Conditionnement

Historiquement, la spécification de l'algorithme utilisé dans la méthodologie AAA était très proche de celle du langage synchrone *Signal*, la spécification du conditionnement en était directement héritée. Après une rapide présentation de la spécification du conditionnement en Signal, nous présenterons la spécification du conditionnement dans le format DC (format commun des langages synchrones) puisque nous avons choisi une modélisation compatible avec ce format.

2.4.1 Conditionnement en Signal

Définitions

Dans ce langage, le conditionnement est exprimé par les processus élémentaires `when` et `default` , modélisés dans le graphe d'algorithme par deux sommets dont voici une brève définition :

Note : dans les d´efinitions qui suivent, A est un signal (suite semi-infinie de valeur) de type quelconque qui peut ˆetre pr´esent ou absent, B est un signal de type bool´een qui peut ˆetre absent, pr´esent de valeur vraie,

pr´esent de valeur fausse. Pour ces signaux nous pouvons d´efinir $P(A)$ (resp. $P(B)$) l'horloge des instants où le signal A (resp. B) est pr´esent. Pour les signaux de type bool´een on d´efinit l'horloge $T(B)$ qui est l'horloge des instants où B est pr´esent et vrai. $T(B)$ est inclus dans $P(B)$ ($P(B) \subset T(B)$).

- When permet le sous-échantillonnage : C = A `when` B, indique que C (de même type que A) prend la valeur de A quand A est présent et que le signal B est aussi présent et sa valeur est vraie. L'horloge de C est alors définie par $P(A) \bigcap T(B)$,

- Default permet le mélange de signaux : C = A `default` B, indique que C prend la valeur de A si A est présent (que B soit présent ou non), sinon C prend la valeur de B si B est présent et (donc A absent), l'horloge de C est définie alors par $P(A) \bigcup P(B)$,

Il est important de souligner que dans ce langage, comme chaque signal ne possède une valeur qu'aux instants logiques où il est présent, si l'on veut utiliser sa valeur à un instant différent de l'instant qui l'a produit il faut le mémoriser à travers un `CELL_WHEN` (composé d'un `DEFAULT`, d'un retard et d'un WHEN). Son horloge (dite libre) est définie comme l'intersection de l'horloge d'écriture en amont et de l'horloge de lecture en aval. Pour fixer l'horloge libre d'un signal A, il faut utiliser la directive de compilation `SYNCHRO(A, B)` qui impose alors que l'horloge de A prenne celle de B ($P(A) = P(B)$).

Conversion Signal-AAA

La transformation d'un graphe signal en graphe d'algorithme AAA conserve la sémantique des sommets `when` et `default`. Cependant, dans le but de générer un code exécutable, il est nécessaire de spécifier explicitement l'horloge de chaque signal afin de déterminer quand et comment exécuter chacun des sommets du graphe. Ce calcul d'horloge, non trivial, est déjà effectué par le compilateur Signal, ses résultats (ce sont des `SYNCHRO`) sont spécifiés dans le graphe d'algorithme AAA à l'aide d'arcs spéciaux nommés *exec* et *execroot* :

– *execroot* est un hyperarc non orienté qui connecte tous les sommets exécutés inconditionnellement (lors de chaque itération du motif),

– *exec* est un hyperarc orienté dont l'unique sommet source est une opération qui produit un signal booléen dont la valeur conditionne l'exécution des sommets puits.

Exemple 2.4.1 *La figure 2.5 représente un graphe d'algorithme dans la spécification Signal-AAA qui effectue le calcul de la valeur absolue d'un nombre.*

L'opération IN est connectée à $LESS$ qui produit un booléen de valeur `TRUE` si la valeur de son entrée est inférieure à zéro, et `FALSE` sinon. Ce booléen est connecté à une opération $W1$ (opération when) dont la sortie est connectée à une opération NEG qui produit une sortie dont la valeur est l'inverse de la valeur de son entrée. Cette sortie est connectée à $D1$ (opération default) dont le rôle est de recopier en sortie la valeur inversée par NEG si elle existe et sinon de prendre directement la valeur issue de IN.

Comme les opérations $W1$ et NEG sont exécutés conditionnellement selon la valeur de l'opération $LESS$, il faut le spécifier par l'intermédiaire d'un arc `exec` entre $LESS$, $W1$ et NEG, toutes les autres opérations sont exécutées inconditionnellement.

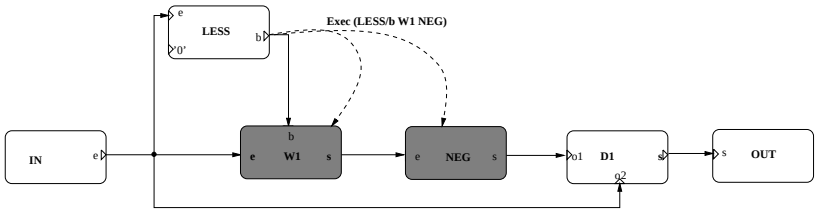


FIG. 2.5: *Calcul valeur absolue avec when default et exec*

Conclusion

La spécification d'algorithme conditionné nécessite donc l'utilisation de sommets spécifiques *when* et *default,* mais aussi des hyperarcs spécifiques `exec` et `execroot` . Ceci permet une compatibilité directe avec le langage Signal, la transformation de graphe peut être automatisée.

Lorsque le graphe n'est pas issu de Signal, la compatibilité directe avec la sémantique Signal entraîne un surcoût de spécification. En effet, il faut exprimer le conditionnement avec des sommets `when` et `default` ainsi qu'avec les hyperarcs `exec` et `execroot` . Or, on remarque que l'arc connecté au `when` est souvent connecté au même sommet que l'arc `exec` (Cf. figure 2.5). C'est pourquoi nous avons donc choisi de proposer une spécification unique du conditionnement.

Remarque 14 *D'autre part la nécessité en Signal de créer des* `CELL WHEN` *pour mémoriser les signaux (puisqu'ils n'existent qu'à des instants logiques précis) ne reflète pas la réalité de l'implantation. Dans le cas d'une mémoire, par exemple, la valeur que l'on y a écrite est conservée jusqu'à la prochaine écriture, l'horloge de lecture peut avoir une fréquence inférieure à celle d'écriture sans pour autant nécessiter l'adjonction d'éléments* `CELL WHEN` *dans la description. Pour générer un code mieux optimisé il serait préférable de supprimer ces éléments de la description.*

2.4.2 Conditionnement DC

Pour simplifier la spécification du conditionnement, mais aussi pour élargir la compatibilité de notre modèle à tous les langages synchrones, nous avons choisi de spécifier le conditionnement tel qu'il est dans le format commun des langages déclaratifs synchrones, DC (Declarative Code [44]), basé sur la notion de *condition d'activation.*

Principes

C'est un format de haut niveau dédié à la description de programmes déclaratifs synchrones flots de données. Un programme DC décrit une machine réactive, la conduite de cette machine dans le temps correspond à une séquence infinie de réactions. Un programme DC gère des objets typés de type flots de données, qui à chaque instant de la vie du programme, détiennent une valeur. La définition d'un flot spécifie sa valeur courante comme une fonction des valeurs courantes ou passées d'autres flots. Un flot est mis à jour quand sa condition d'activation est vraie : chaque définition de flot comporte donc une condition d'activation correspondant à un flot de type booléen.

Si cette condition est fausse, le flot :

- est non défini si la condition n'a jamais eu lieu avant et qu'il n'a pas de valeur donnée par défaut,

- prend la valeur définie par défaut si elle existe et si la condition n'a encore jamais été vraie,
- prend la dernière valeur définie quand sa condition d'activation était vraie.

Par rapport à Signal, la notion de présence et d'absence d'un signal n'existe plus car sa valeur est rémanente.

Concrètement, un programme DC est décrit par un ensemble de table (table de flots, table noeuds, table de fonctions), auxquelles on fait référence à l'aide d'indices. Il existe aussi des flots prédéfinis :

– flots booléen toujours faux : $@0,

– flots booléen toujours vrais : $@1.

En DC la condition d'activation est représentée par le mot "at", d'autre part dans DC les arguments des opérations (affectations..) se font par références dans des tables de déclarations, les opérations elles-mêmes sont définies par des références dans des tables.

Exemple 2.4.2 *La définition suivante (son index vaut 0) indique que le flot d'index numéro 4 (défini dans la table des flots) reçoit la valeur du flot d'index 3 (qui doit être de même type) quand la condition d'activation d'index 5 (flot de type booléen) est vraie. Cette affectation s'effectue donc à l'horloge $T(Flot5)$, soit l'horloge d'écriture du flot 2 : $H_{Write}(Flot4) = T(Flot5)$, et on pourra vérifier que la fréquence de l'horloge de lecture du flot 4 reste inférieure ou égale à celle de son écriture : $H_{Read}(Flot4) < H_{Write}(flot4) = T(Flot5)$*

```
0: equ:  4 3 at:  5
```

La définition suivante (d'indice 1) indique que le flot d'indice 3 est égal au résultat de la fonction d'indice 7 (dans la table de définition des fonctions) quand le flot (condition d'activation) d'indice 5 est vrai. La fonction d'indice 7 prend 2 flots en argument, d'indices respectifs 8 et 9 :

```
1: equ:  3 7(8,9)  at:  5
```

En DC il existe 44 fonctions prédéfinies pour effectuer les opérations courantes telles que les opérations arithmétiques et logiques. Pour les utiliser il faut utiliser la syntaxe $IndiceFonction.

Exemple 2.4.3 *Pour copier la somme (fonction 13) des flots d'entiers d'indice 1 et 2 dans le flot d'indice 3 lorsque le flot d'indice 4 est vrai, il faut écrire :*

```
2: equ:  3 $13(1,2)   at:  3
```

La ligne suivante indique qu'a chaque instant la valeur du flot d'indice 3 est égale à la somme des valeurs des flots d'indice 1 et 2.

```
3: equ:  3 $13(1,2)   at:  $@1
```

Parmi les fonctions prédéfinies on trouve une fonction particulière qui est la fonction $conditionnelle$. Cette fonction possède 3 arguments $c, e1, e2$ et une sortie s : c doit être connecté à un flot booléen. Si ce

dernier est vrai, le flot de sortie s est égal au flot $e1$, sinon il est égal au flot $e2$.

Exemple 2.4.4 *Le flot 8 est égale au flot 2 quand le flot 1 est vrai, sinon le flot 8 est égal au flot 3. Cette égalité est vrai quand la condition d'activation d'indice 4 est vrai.*

```
4: equ 8 $0(1,2,3)   at: 4
```

Remarque 15 *La sémantique adopté en DC, est plus proche d'une implantation sur machine programmable que celle de Signal. En effet le signal garde sa valeur jusqu'à sa prochaine modification. Il n'est donc plus nécessaire comme en Signal, d'ajouter des* `CELL` *chaque fois que l'horloge d'écriture du signal est plus élevée que celle de lecture.*

2.4.3 Conditionnement AAA

Formalisation

Chaque condition d'activation est modélisée par un arc de dépendance dit *de conditionnement* [102] qui induit une condition d'activation sur chaque opération réceptrice d'un tel arc. Les opérations puits de ces arcs sont dites conditionnées, elles ne sont exécutables que si leur condition d'activation est vraie et que leurs autres données sont présentes.

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.

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

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

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

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

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

Soit η^{-1}, l'application qui, à chaque opération associe les booléens et leurs complémentaires qu'elle émet.

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

Opérations exclusives

Dans le cas où des opérations sont exécutées de manière exclusive (jamais exécutées dans la même itération), mais qu'une opération prend en entrée les données produites par l'une ou l'autre des opérations exclusives, il faut insérer un sommet de type `default` en signal, ou `conditionnelle` en DC, entre cette opération et les opérations exécutées exclusivement. L'inconvénient de ces sommets est de n'avoir que deux entrées : dans la description d'un automate par exemple, les répercussions sur la complexité graphique et les possibilités d'optimisation sont relativement importantes. En effet, si il est possible de prévoir que des opérations ne sont jamais exécutées dans la même itération, les tampons nécessaires pour stocker les données qu'elles produisent ne sont pas utilisés pendant cette itération. Nous avons donc introduit un sommet `merge` à n entrées, c'est une extension de la conditionnelle limitée à deux entrées de DC.

Représentation graphique

Les dépendances de données doivent être différenciées graphiquement des dépendances de conditionnement. C'est pourquoi nous représenterons les dépendances de conditionnement par des arcs pointillés.

Exemple 2.4.5 *Le graphe de la figure 2.6 a) modélise un automate simple à trois états et une sortie avec des* `when` *et des* `default` *à deux entrées, la figure b) modélise le même automate mais avec des conditions d'activation et des* `Merges` .

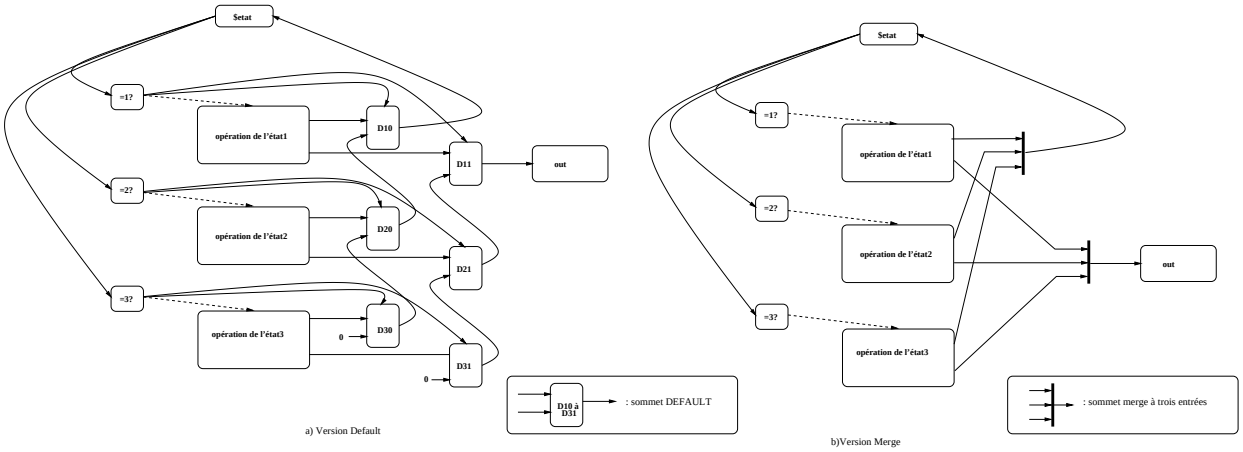


FIG. 2.6: *Graphe d'un automate*

Exemple 2.4.6 *La figure 2.7 représente le graphe de l'algorithme de calcul de la valeur absolue obtenue en utilisant ces nouvelles définitions.*

2.5 Étiquetage pour génération d'exécutif

Le formalisme présenté jusqu'ici pour modéliser les algorithmes, est suffisant pour travailler sur les problèmes de distribution et d'ordonnancement. Cependant, dans cette thèse nous allons jusqu'à la phase finale d'implantation de l'algorithme sur l'architecture. Cette dernière est obtenue par transformation des

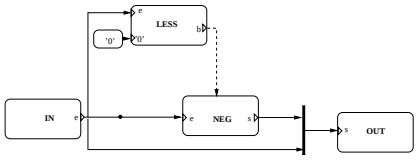


FIG. 2.7: *Valeur absolue avec horloge d'activation*

graphes d'architecture et d'algorithme en un graphe d'implantation (Cf. chapitre 3), lui-même transformé ensuite pour générer automatiquement un exécutif exécutable par les composants de l'architecture (Cf. Partie II). Cette génération de code requiert un certain nombre d'informations qui nécessitent un enrichissement du modèle d'algorithme présenté jusqu'ici.

Nous avons vu que chaque sommet correspond à une opération de calcul ou d'entrée-sortie, les dépendances de données correspondent aux données transmises entre les opérations. Plusieurs dépendances de données peuvent aboutir à un même sommet (consommateur) ou être issues d'un même sommet (producteur). Nous allons voir que cette description n'est pas suffisante pour générer automatiquement un exécutif (chapitre 7), ni pour effectuer les optimisations du chapitre 4. En effet, si chaque sommet du graphe correspond à l'appel d'une fonction compilée séparément, et chaque dépendance à un tampon mémoire, il faut être capable de générer l'appel de la fonction en transmettant les tampons dans l'ordre qui correspond à celui attendu par la fonction. Tel que formalisé précédemment, le modèle de graphe d'algorithme ne permet pas de stocker l'information concernant l'ordre des arguments. De plus, lors de la génération de code, mais aussi pour la phase d'optimisation, il est indispensable de connaître le type d'une dépendance, c'est à dire la quantité de données associée à chaque dépendance du graphe. Lors de l'optimisation, (chapitre 4), cette information est pourtant nécessaire pour calculer la durée de transfert de ces données (la durée de transfert est une fonction de la quantité de données dans notre modèle), mais aussi pour vérifier que la mémoire disponible dans l'architecture est suffisante.

Pour optimiser et générer automatiquement l'exécutif, nous proposons de valuer chaque dépendance de donnée par un quadruplet $(pos\_out, type, taille, pos\_in)$.

- $type$ et $taille$ sont des valeurs entières permettant d'associer une quantité (égale à $type * taille$) de bits à chaque dépendance de données,
- pos_out et pos_in indiquent la position du tampon correspondant dans la liste d'arguments des fonctions productrice et consommatrice(s) de ce tampon. Pour leur codage, nous avons eu une première idée qu'il est intéressant d'étudier pour justifier la complexité introduite dans la seconde solution qui a été retenue.

2.5.1 Première proposition : codage direct

Chaque arc étant réalisé par un tampon mémoire lors de l'implantation, on définit pos_out comme la position de ce tampon dans la liste d'arguments de la fonction correspondante au sommet source de cette dépendance. Inversement, pos_in est la position de ce tampon dans la liste d'arguments de la fonction correspondant au sommet puits de cette dépendance.

Exemple 2.5.1 *Dans le graphe de la figure 2.8, le sommet A est connecté à deux arcs, c'est à dire qu'il produit deux données distinctes : un scalaire (dépendance d1) et un tableau de 10 éléments (dépendance d2). Le scalaire est consommé par le sommet B qui à son tour produit un autre scalaire (dépendance d3). Le sommet C consomme le tableau produit par A et le scalaire produit par B.*

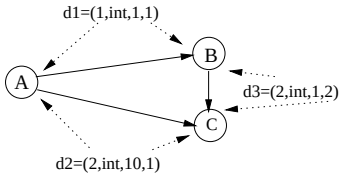


FIG. 2.8: *Graphe d'algorithme valué*

Le premier élément du quadruplet (1,int,1,1) associé à l'arc $d1$ indique que le tampon correspondant à $d1$ est passé en premier dans la liste d'appel à la fonction correspondant au sommet A. De même, on constate sur cet exemple que le tampon correspondant à l'arc $d2$ est passé en deuxième argument. Si le sommet A est associé à une fonction A, l'implantation monoprocesseur en langage C de l'appel à la fonction A pourrait être `''A(d1,d2);''` *(en ayant préalablement déclaré les tampons $d1$ et $d2$ par* `''int d1;''` *et* `''int d2[10];''` *). Selon le même principe, le code généré pour les deux autres sommets serait* `''B(d1,d3);''` *et* `''C(d2,D3);''` *.*

Si le graphe d'algorithme n'était pas basé sur des hyperarcs mais uniquement sur des arcs, ce principe serait suffisant pour la génération automatique de code. Dans le cas des hyperarcs, il pose le problème suivant : un hyperarc connecte une fonction source à plusieurs fonctions puits. Lors de la génération de code, cet hyperarc correspond à un tampon qu'il faudra donner en argument à chaque fonction puits. La position de ce tampon dans la liste d'appel des fonctions puits est donnée par la valeur du quatrième élément du quadruplet. Comme cette valeur est unique (un seul quadruplet par hyperarc), la position de ce tampon est nécessairement identique dans l'appel de chaque fonction puits.

Ainsi il est parfois impossible de valuer les arcs tels que la liste d'arguments de chaque fonction commence à l'indice 1, la figure 2.9 en présente un exemple. Sur ce graphe, il est impossible de trouver des quadruplets tels que les arguments d'une fonction soient tous placés à partir de l'indice 1. Dans cet exemple le code généré pour les sommets E et F serait `E(?,d2)` et `F(?,?,d3)` : la fonction E ne doit pas avoir de premier argument, la fonction F n'a qu'un troisième argument.

D'autre part, on constate que selon ce principe, l'ordre des arguments dans la liste d'arguments de chaque fonction est totalement dépendant des connexions dans le graphe de l'algorithme.

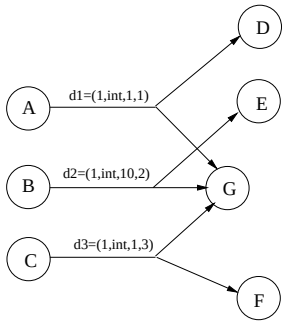


FIG. 2.9: *Problèmes dans le cas des hyperarcs*

2.5.2 Seconde proposition : table d'indirection

Pour éviter les inconvénients cités précédemment, nous associons une table "d'indirection" à chaque opération du graphe d'algorithme. Cette table indique, pour chaque valeur d'un quadruplet, la position correspondante du tampon dans la liste d'arguments de la fonction. Les valeurs pos_out et pos_in peuvent maintenant valuer les hyperarcs sans tenir compte de la liste d'arguments réels des fonctions puits (Cf. phase b) figure 2.10. Par contre, en attribuant ces valeurs, il est important de vérifier que les valeurs des positions sont bien différentes pour tous les arcs connectés à un même sommet (cela peut nécessiter de ne pas utiliser certaines valeurs). Pour cela la table d'indirection doit être construite après construction du graphe complet de l'algorithme et de la valuation des hyperarcs.

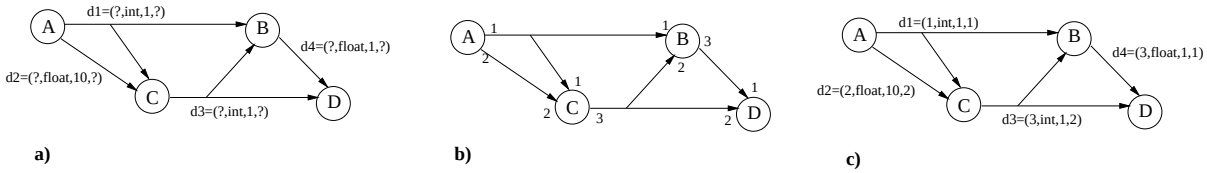


FIG. 2.10: *Étapes de construction des quadruplets*

Exemple de table pour F et E :

Valeur pos_in/pos_out	position réel
3	arg numero 1

Valeur pos_in/pos_out	position réel
2	arg numéro 1

etc

Soit $type(d'_i)$ le type d'une dépendance de données d_i, appartenant à un ensemble prédéfini de type (cette valeur correspond au second élément du quadruplet qui est associé à d'_i). Soit $q(d'_i)$ la quantité de données associées à cette dépendance (cette valeur correspond au troisième élément du quadruplet qui est associé à d'_i). Soit $Pos_{in}(d'_i)$ la position du tampon (modélisé par la dépendance de données) dans la liste d'appel du producteur ($\gamma^{-1}(d'_i)$) de la dépendance. Soit $Pos_{out}(d'_i)$ la position du tampon (modélisé par la dépendance de données) dans la liste d'appel du consommateur ($\gamma(d'_i)$) de la dépendance.

Chapitre 3

Modèle d'implantation

Sommaire

A partir d'un graphe d'algorithme et d'un graphe d'architecture, il est possible de construire l'ensemble des graphes d'implantation dits valides au moyen de la composition de trois relations (*le routage, la distribution* et *l'ordonnancement*) comme cela est formalisé dans [102]. Le modèle d'architecture utilisé dans cette thèse étant moins fin que celui présenté dans le premier chapitre (prise en compte des mémoires, des arbitrages, extensions des modèles de communications), nous allons ici enrichir ces différentes relations qui font partie du modèle d'implantation. Ainsi nous allons pouvoir décrire finement l'allocation de la mémoire (mémoire programme, mémoire données . . .), mais aussi tous les types de communications possibles entre opérateurs (communications par passages de messages, par mémoires partagées), tout en tirant profit des caractéristiques particulières des types de sommets utilisés (SAM point à point, multipoint avec support matériel ou non du broadcast).

Pour cela nous commençons, dans une première partie, par présenter l'ancien modèle d'architecture utilisé dans [102] et les différentes relations qui conduisent à l'ensemble des graphes d'implantation valides. Dans une seconde partie, nous enrichirons ces relations de façon à prendre en compte notre nouveau modèle d'architecture tout en donnant quelques exemples relatifs à ces nouvelles relations.

3.1 Précédent modèle

Deux modèles d'architecture hétérogène sont présentées dans [102], le modèle encapsulé et le modèle développé. Le second est plus précis que le premier, mais ils sont tous deux limités aux communications inter-processeurs par média SAM.

L'hypothèse est faite que chaque opération exécutée accède uniquement à la RAM connectée à l'opérateur qui l'exécute. Les opérations ne font jamais d'accès direct à la SAM inter-processeurs, même si elles doivent communiquer avec des opérations exécutées par d'autres opérateurs. Nous verrons dans la section consacrée aux communications que les transferts inter-opérateurs sont alors réalisés par des opérations de communication (Cf. § 3.1.4). Cette hypothèse offre deux avantages. Tout d'abord, cela permet d'éviter que les exécutions des opérations soient couplées entre elles par l'accès séquentiel dans les SAM (chaque écriture impliquant une lecture des données précédemment écrite). Tout couplage rend en effet inutilisable la caractérisation individuelle de la durée des opérations, nécessaire pour effectuer des optimisations (Cf. 4.1). De plus, sans cette hypothèse, le codage de chaque opération dépend du type d'accès des RAM ou des SAM, dans lesquels elle doit lire ou écrire n arguments , il faudrait 2^n codages différents de ces opérations afin de pouvoir les distribuer sur n'importe quel opérateur de l'architecture. Avec cette hypothèse, où tous les arguments sont accédés dans de la RAM, un unique codage suffit. Le modèle d'algorithme utilisé ici correspond au modèle d'algorithme décrit dans le précédent chapitre.

3.1.1 Modèle d'architecture

Un graphe d'architecture encapsulé correspond à un couple (P, H) où P représente les processeurs (chaque processeur encapsule l'ensemble des unités d'un processeur du modèle développé exposé ci-dessous) et H les liaisons physiques de type SAM entre ces processeurs. L'architecture peut être hétérogène, les opérateurs et les liaisons physiques peuvent être de types différents, la durée d'exécution de chaque opération dépend du type de l'opérateur qui l'exécute de même que la durée de chaque communication inter-processeurs dépend du type de la liaison qui la supporte.

Un graphe d'architecture développé G_{ar} d'une architecture composée de P processeurs est décrit par un couple (S, H_d) où S est l'ensemble des sommets opérateurs et H_d l'ensemble des hyperarcs reliant des opérateurs. S est composé de deux types de sommet opérateur, les opérateurs de calcul S_{calc} (un sommet par processeur) et les opérateurs de communication S_{com} (des communicateurs). Ainsi, chaque processeur p du modèle encapsulé est composé d'un sommet opérateur de calcul $S_{p,cal}$ et d'un ou plusieurs opérateurs de communication S_{p,com^j} connectés par un hyperarc intra-processeur modélisant une RAM unique. L'ensemble de ces hyperarcs,noté C, est un sous-ensemble de H_d. On a $S_{cal} = \bigcup_{p \in P} S_{p,cal}$ et $S_{com} = \bigcup_{p \in P} (\bigcup_j S_{p,com^j})$. Les opérateurs de communication appartenant à des processeurs différents sont connectés par des hyperarcs qui modélisent chacun une SAM. L'ensemble de ces hyperarcs, noté H, est un sous-ensemble de H_d. On appelle *média* m_j du graphe d'architecture encapsulé, un hyperarc $h_j \in H$ et les opérateurs de communication correspondants appartenant à des processeurs différents.

Par la suite nous ne nous intéressons qu'à l'enrichissement du modèle le plus précis, c'est à dire celui du modèle développé.

3.1.2 Routage

De nombreux travaux traitant de l'implantation d'algorithmes sur des machines parallèles posent comme hypothèse que l'architecture parallèle est complètement connexe, ils supposent donc l'existence d'un média de communication joignant chaque paire de processeurs de la machine. Comme ce n'est pas nécessairement le cas du graphe d'architecture présenté plus haut. Une relation $\mathcal{R}_{routage}$ est introduite, appliquée au graphe d'architecture, elle construit des chemins (combinaisons de média) dans le graphe d'architecture, de façon

à pouvoir faire communiquer tous les couples opérateurs de calcul du graphe.

$$\begin{array}{ccc} G_{ar} & \stackrel{\mathcal{R}_{routage}}{\longrightarrow} & G'_{ar} \\ (S, H_d) & \stackrel{\mathcal{R}_{routage}}{\longrightarrow} & (S, R_d) \end{array}$$

3.1.3 Distribution

Après cette étape préliminaire qu'est le routage et qui s'applique uniquement au graphe d'architecture, une allocation spatiale du graphe de l'algorithme sur le graphe de l'architecture est réalisée. L'allocation spatiale, appelée ici *distribution*, se décompose en deux grandes étapes, *le partitionnement* et *la communication.*

3.1.3.1 Partitionnement

La première transformation consiste à partitionner le graphe d'algorithme en n sous-graphes disjoints qui seront chacun associé à un processeur, c'est à dire à un sommet opérateur d'un processeur. Le cardinal de chaque partition doit être inférieur ou égal au nombre d'opérateurs. Tous les opérateurs de calcul du graphe de l'architecture sont supposés capables d'exécuter toutes les opérations du graphe d'algorithme.

Soit Π', l'application qui, à chaque opération de calcul, associe l'opérateur de calcul sur lequel elle est distribuée :

$$\begin{array}{rccl} \Pi' : & O' & \rightarrow & S_{cal} \\ & o'_i & \mapsto & \Pi'(o'_i) = p_j \end{array}$$

Soit Π'^{-1}, l'application réciproque de Π', qui associe à chaque opérateur, l'ensemble des opérations de calcul distribuées sur cet opérateur :

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

Ainsi, $\Pi'^{-1}(p)$ correspond à l'ensemble, noté O'_p, des opérations distribuées sur l'opérateur p. Ce partitionnement donne lieu à deux types de dépendances de données :

- les dépendances de données intra-partition ${D'}_p$ (appelées aussi intra-processeur) sont définies par :

$$D'_p \subseteq O'_p \times \mathcal{P}(O'_p) \ \ \forall p \in S_{cal}$$

  On note ${D'}_P$ (avec ${D'}_P \subseteq D'$), l'ensemble des dépendances intra-processeur de G_{al}. Dans [102] il est démontré que ${D'}_p$ définit un ordre partiel, noté $\preceq_P$ sur l'ensemble des opérations de O',

- les dépendances de données inter-partitions (appelées aussi inter-processeurs), notées ${D'}_r$, elles correspondent aux dépendances de données entre opérations distribuées sur des opérateurs différents. Elles vont donner lieu à un transfert de données sur chaque média qui compose la route r joignant les opérateurs émetteur et récepteur.

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

  On note D'_R l'ensemble des dépendances entre opérations dépendantes et distribuées sur des opérateurs différents. Dans [102] il est montré que D'_R définit un ordre partiel $\preceq_R$ sur l'ensemble des opérations O'.

A partir d'un graphe d'algorithme et d'un graphe d'architecture routé, le partitionnement permet de construire les graphes partitionnés $G_{partR}(.)$:

$$\begin{array}{rcl} (G_{al}, G'_{ar}) & \stackrel{\mathcal{R}_{Part}}{\longrightarrow} & (G_{partR}, G'_{ar})(.) \\ ((O', D'), (S, Rd)) & \stackrel{\mathcal{R}_{Part}}{\longrightarrow} & (\bigcup_{p \in S_{cal}} (O'_p, D'_p), \bigcup_{r \in R} D'_r) \end{array}$$

Il a été démontré dans [102] que l'ordre partiel, $\preceq$, de l'algorithme est conservé dans le graphe partitionné :

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

3.1.3.2 Communication

Les dépendances de données inter-processeurs ($D_R{}'$) ont été associées à des routes faites de combinaisons de média. Chaque dépendance de données donne lieu à un transfert de données sur chacun des média composant la route reliant les deux unités de calcul. Comme ces média sont constitués d'un ensemble de sommets opérateurs de communication connectés, chaque transfert de données sur un média est modélisé par un couple d'*op´erations de communication.* Pour chaque dépendance de données inter-processeurs, on insère entre les deux opérations de calcul dépendantes autant de couple d'opérations de communication qu'il y a de couple d'unités de communication utilisées pour ce transfert sur les média qui composent la route. Soit :

- O'' l'ensemble des opérations de communications ajoutées au graphe de l'algorithme. Soit O l'ensemble des opérations de calcul et de communication du graphe de l'algorithme, on a $O = O' \cup O''$,
- O''_p l'ensemble des opérations de communication distribuées sur les unités de communication du processeur p,
- D'' l'ensemble des arcs entre opérations de communication distribuées sur des opérateurs de communication appartenant à des processeurs différents, il induit un ordre partiel $\preceq_h$:

$$D'' = \bigcup_{p,q} D''_{p,q \in P} \quad avec \quad D''_{p,q} \subseteq O''_p \times O''_q \; p \neq q$$

- D^*_P, l'ensemble des arcs entre opérations de calcul et de communication et entre opérations de communication, distribuées sur le même processeur. D^*_p sera distribué sur la RAM (hyperarc de C) de ce processeur p. D^*_P induit un ordre partiel $\preceq_c$:

$$D^*_P = \bigcup_{p \in P} D^*_p \quad avec \quad D^*_p \subseteq (O'_p \times O''_p) \cup (O''_p \times O''_p) \cup (O''_p \times O'_p)$$

A partir d'un graphe partitionné, la relation appelée communication, notée $\mathcal{R}_{com}$ conduit à un ensemble de graphes $G_{comR}(.)$:

$$\begin{array}{rcl} (G_{par\,R}, G'_{ar}) & \stackrel{\mathcal{R}_{com}}{\longrightarrow} & (G_{comR}, G'_{ar})(.) \\ (\bigcup_{p \in P} (O'_p, D'_p), \bigcup_{r \in R} D'_r) & \stackrel{\mathcal{R}_{com}}{\longrightarrow} & ((\bigcup_{p \in P} O'_p) \cup (\bigcup_{p \in P} O''_p), D'_P \cup D'' \cup D^*_P) \end{array}$$

Dans [102] il est démontré qu'à l'issue de la communication, l'ordre partiel associé à ces graphes est inclus dans l'ordre partiel initial du graphe d'algorithme :

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

3.1.4 Ordonnancement

Les opérateurs de calculs et les opérateurs de communication sont des machines séquentielles à états finis, ils ont à exécuter chacun un sous-graphe qui doit être un ordre total. Dans ce chapitre nous allons donc étudier la relation appelée *ordonnancement*, notée $\mathcal{R}_{ordo}$ qui permet de renforcer l'ordre partiel ($\preceq$) en un ordre total ($\prec$) entre les opérations de chaque partition. Cette relation correspond à l'allocation temporelle du graphe distribué, sur le graphe de l'architecture.

- on renforce donc l'ordre partiel D'_p ($\preceq_p$) de chaque partition associée à un opérateur de calcul d'un processeur p, en un ordre total $\bar{D}'_p$ ($\prec_p$) à l'aide d'arcs de dépendances d'exécution sans données, appelées *arcs de pr´ec´edence*), notés D'''_p, entre les opérations O' du graphe d'algorithme ($\bar{D}'_p = D'_p \cup D'''_p$),

- de même on renforce l'ordre partiel $D''_{p,c}$ ($\preceq_{p,c}$) de chaque opérateur de communication c de chaque processeur p en un ordre total $\bar{D}''_{p,c}$ ($\prec_{p,c}$) à l'aide de précédences $D'''_{p,c}$ entre les opérations de communication O'' ($\bar{D}_{p,c} = D''_{p,c} \cup D'''_{p,c}$),

  Remarque 16 *Nous verrons lors de la section 3.2.4 que l'ordonnancement des opérations de communication doit respecter certaines règles afin d'éviter tout interblocage dû à l'accès séquentiel des mémoires SAM.*

- l'ordre partiel $\preceq_c$ donné par les dépendances de données $D_p{}^*$ est inchangé.

Après ordonnancement, l'ensemble des dépendances du graphe d'algorithme distribué et ordonnancé (nous l'appellerons graphe d'implantation) est défini par :

$$D = (\bigcup_{p \in S_{cal}} \bar{D}'_p) \cup (\bigcup_{p \in S_{cal}, c \in S_{com}} \bar{D}''_{p,c}) \cup D^*_P$$

L'ordre partiel du graphe d'implantation est défini par :

$$\trianglelefteq = (\bigcup_{p \in s_{cal}} \prec_p) \cup (\bigcup_{p \in S_{cal}, c \in S_{com}} \prec_{p,c}) \cup (\bigcup_{c \in C} \preceq_c)$$

A partir d'un graphe G_{comR} issu de la communication, la relation d'ordonnancement permet de construire l'ensemble des graphes ordonnancés $G_{ordo}(.)$:

$$\begin{array}{rcl} (G_{comR}, G'_{ar}) & \stackrel{\mathcal{R}_{Ordo}}{\longrightarrow} & (G_{ordoR}, G'_{ar})(.) \\ ((\bigcup_{p \in P} O'_p) \cup (\bigcup_{p \in P} O''_p), D'_P \cup D'' \cup D^*_P) & \stackrel{\mathcal{R}_{Ordo}}{\longrightarrow} & ((\bigcup_{p \in P} O'_p) \cup (\bigcup_{p \in P} O''_p), \bar{D}_P \cup \bar{D}'' \cup D_p{}^*) \end{array}$$

3.2 Enrichissement du modèle

Le nouveau modèle d'architecture présenté dans le premier chapitre étant beaucoup plus fin et précis que celui utilisé dans [102], nous devons enrichir le modèle d'implantation et plus précisément les relations de routage, distribution (partitionnement et communication) et ordonnancement conformement à notre nouveau modèle d'architecture. Ceci nous permettra d'effectuer un plus grand nombre d'optimisation (mémoires, communications, etc) comme nous le verrons dans le prochain chapitre.

3.2.1 Modèle d'architecture

L'hypothèse faite dans [102] concernant l'accès uniquement aux RAM par les opérations (Cf. 3.1) est conservé ici. Comme précédemment, nous choisissons donc de coder les opérations de façon à ce qu'elles lisent et écrivent uniquement dans les mémoires RAM connectées aux opérateurs qui les exécutent. Ainsi, les caractérisations individuelles des durées des opérations restent utilisables lors de l'optimisation (Cf. chapitre suivant), et le codage de chaque opération ne dépend pas de son implantation. Ce choix a une conséquence sur notre modèle d'architecture auquel il faut ajouter une règle supplémentaire :

Règle 7 *Chaque opérateur du graphe de l'algorithme doit être connecté à au moins une RAM.*

Remarque 17 *Bien que restrictive, cette règle est acceptable dans la mesure où il est très rare qu'un opérateur faisant partie d'un processeur réel ne soit pas connecté à une mémoire RAM.*

Remarque 18 *Que la mémoire RAM soit partagée au non, le codage des opérations reste unique, c'est seulement l'allocation des données qui diffère au moment de la génération d'exécutif, comme nous le verrons lors de la seconde partie.*

Dans le modèle d'architecture présenté au début de cette thèse, un processeur peut être connecté à plusieurs mémoires RAM (représentées par des sommets) et non plus à une seule mémoire (représentée par un hyperarc joignant les opérateurs de communication appelés ici communicateurs). Des sommets Bus/Mux/Demux et Bus/Mux/Demux/Arbitre ont été introduits pour inter-connecter les opérateurs et communicateurs aux mémoires. Les communicateurs ne sont plus uniquement connectés par une SAM (modélisée par un hyperarc joignant les communicateurs), mais ils peuvent aussi être connectés par une RAM. Le tableau suivant (Cf. figure 3.1) illustre les correspondances et les différences entre le modèle d'architecture développé présenté dans [102] (noté modèle AV) puisque c'est le plus précis des anciens modèles, et notre nouveau modèle d'architecture présenté dans le premier chapitre (noté nouveau modèle). Les éléments équivalents des deux modèles sont placés sur la même ligne :

G_{ar}	Mod`ele AV	Nouveau mod`ele
Sommets	S_{calc} S_{com}	S_{opr} S_{com} S_{RAM} S_{SAM} S_{bus}
Arcs Arcs	 C (RAM) H (SAM)	S

FIG. 3.1: *Comparaison des modèles d'architecture*

Nous constatons qu'il existe de nouveaux sommets auxquels nous allons associer de nouvelles opérations ajoutées dans le graphe d'algorithme.

3.2.2 Routage

Le routage ne consiste plus à construire des routes uniquement composées de sommets communicateurs. Chaque route est maintenant composée de sommets communicateurs, de sommets RAM ou SAM et de sommets Bus/Mux/Demux et Bus/Mux/Demux/Arbitre. Nous distinguons maintenant deux grands types de routes :

- l'ensemble des routes iso-opérateur R_{iso}, qui ont un même sommet opérateur de départ et d'arrivée, et passent par l'un des sommets mémoire RAM local à cet opérateur,
- l'ensemble des routes inter-opérateurs R_{inter} qui permettent de joindre n'importe quelle paire de sommets opérateurs du graphe de l'architecture.

3.2.2.1 Routes Iso-opérateur

A chaque opérateur p est associé l'ensemble des routes iso-opérateur de p appelé $R_{iso,p}$. Chacune de ces routes est constituée d'une combinaison de sommets Bus/Mux/Demux et d'une RAM. Rappelons que les connexions entre un opérateur et une RAM sont bidirectionnelles (Cf. § 1.2.2.1). Pour un opérateur donné, il existe autant de routes iso-opérateur qu'il y a de RAM connectées (à travers un ou plusieurs Bus/Mux/Demux) à cet opérateur.

Remarque 19 *Étant donné la nature bidirectionnelle des connexions, les routes iso-opérateur correspondent à des cycles dans le graphe de l'architecture.*

Exemple 3.2.1 *La figure 3.2 présente un graphe d'architecture composé de trois mémoires RAM ($R1, R2, R3$) accessibles par un unique opérateur ($Opr1$) à travers deux Bus/Mux/Demux ($b1$ et $b2$). Il est possible de construire les trois routes iso-opérateur suivantes :*

- $r_1 = \{b1, R1, b1\}$
- $r_2 = \{b1, b2, R2, b2, b1\}$
- $r_3 = \{b1, b2, R3, b2, b1\}$

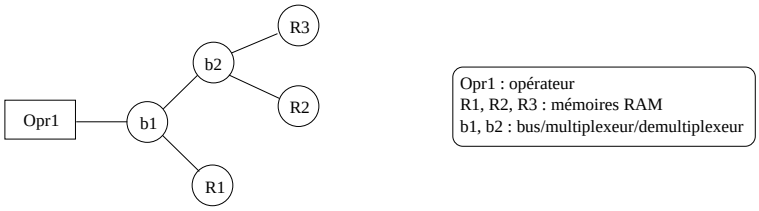


FIG. 3.2: *Routage d'une architecture mono-opérateur*

3.2.2.2 Routes Inter-opérateurs

Chaque route est constituée d'une combinaison de sommets Bus/Mux/Demux, Bus/Mux/Demux/Arbitre, RAM, SAM et d'éventuels communicateurs qui permettent de joindre deux opérateurs p, q, on les note $R_{inter_{p,q}}$

Définition 1 *On définit la longueur d'une route inter-opérateurs par le nombre de sommets RAM ou SAM et de sommets communicateurs qui la composent. La longueur d'une route sera utilisée dans la partie optimisation.*

Remarque 20 *Il est parfois possible de construire plusieurs routes de même longueur, ou de longueurs différentes, entre deux opérateurs. Ces routes sont alors qualifiées de routes parallèles et nous verrons comment équilibrer les communications sur ces routes de façon à minimiser les durées de communication.*

Remarque 21 *Nous verrons dans le chapitre Optimisation que nous ne conservons pas toutes les routes possibles, mais seulement la ou les routes les plus courtes entre chaque couple d'opérateur.*

Nous classons les routes inter-opérateurs en deux catégories : les routes élémentaires, qualifiées aussi de directes, et les routes composées. Les routes composées sont obtenues par assemblages de routes élémentaires.

3.2.2.2.1 Routes élémentaires Les routes élémentaires sont les routes les plus courtes qu'il est possible de construire entre deux opérateurs d'un graphe d'architecture, il en existe trois types qui diffèrent par le type de sommet qui les compose : RAM partagée, RAM partagées et communicateurs, SAM partagée et communicateurs.

- **RAM partagée :** c'est le cas le plus simple, une telle route est constituée d'une RAM partagée par au moins deux opérateurs. Il peut y avoir un ou plusieurs sommets Bus/Mux/Demux entre les opérateurs et la RAM partagée (Cf cas a et b de la figure 3.3). Une route de ce type peut être spécifiée de façon générique par $R = \{s_0, s_1...s_i, r, s_{i+1}, .., s_n\}$, avec $s_1..s_n \in S_{bus}$, $n \geq 0$ le nombre de Bus/Mux/Demux de la route et $r \in S_{RAM}$. Soit R_0 l'ensemble des routes de ce type.

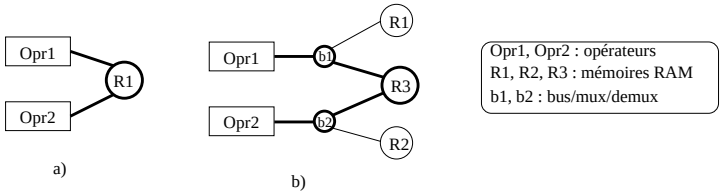


FIG. 3.3: *Routage par RAM partagée (les routes apparaissent en gras)*

- **RAM partagée et communicateurs :** une route élémentaire de ce type peut être construite lorsque deux opérateurs sont chacun connecté à une RAM partagée, elles-mêmes chacune connectées à un communicateur, et que ces communicateurs se partagent une troisième RAM partagée (Cf figure 3.4). A cette route peuvent s'ajouter des Bus/Mux/Demux entre les opérateurs et les RAM, et/ou un Bus/Mux/Demux/Arbitre entre RAM et communicateurs.

  Une route de ce type peut être spécifiée par la séquence : $\{s_1, ...s_i, r_1, s_{b1}, c_1, r, c_2, s_{b2}, r_2, s_{i+1}, ..s_n\}$, avec $n \geq 0$, $s_1...s_n, s_{b1}, s_{b2} \in S_{bus}$, $c_1, c_2 \in S_{com}$, et $r, r_1, r_2 \in S_{RAM}$. Soit R_r l'ensemble des routes de ce type.

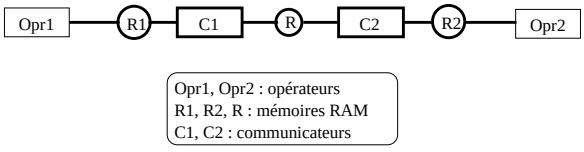


FIG. 3.4: *Routage par RAM et communicateurs (la route apparaît en gras)*

- **SAM et communicateurs :** ces routes diffèrent des précédentes par la présence d'une SAM entre les communicateurs à la place d'une RAM.

  L'ensemble des routes R_s de ce type peut s'écrire : $\{s_1, ...s_i, r_1, s_{b1}, c_1, r, c_2, s_{b2}, r_2, s_{i+1}, ..s_n\}$, avec $n \geq 0$, $s_1...s_n, s_{b1}, s_{b2} \in S_{bus}$, $c_1, c_2 \in S_{opr}$, $r_1, r_2 \in S_{RAM}$ et $r \in S_{SAM}$.

3.2.2.2.2 Routes composées Les routes composées sont construites par combinaison (enchaînement) de plusieurs routes élémentaires. L'enchaînement (ou union "∪") de deux routes élémentaires est réalisé par le partage d'une RAM connectée à l'une des extrémités de chaque route élémentaire. Les données sont acheminées de mémoire en mémoire, jusqu'à atteindre la RAM connectée à l'opérateur destinataire. Il existe trois types de routes composées :

- les routes composées homogènes basées uniquement sur des communications inter-communicateurs par RAM, elles s'écrivent génériquement : $r_1 \cup r_2 \cup \ldots \cup r_n$, avec $r_1 \ldots r_n \in R_r$,
- les routes composées homogènes basées uniquement sur des communications inter-communicateurs par SAM, elles s'écrivent génériquement : $r_1 \cup r_2 \cup \ldots \cup r_n$, avec $r_1 \ldots r_n \in R_s$,
- les routes composées hétérogènes basées à la fois sur des communications inter-communicateurs par RAM et SAM, elles s'écrivent génériquement : $r_1 \cup r_2 \cup \ldots \cup r_n$, où $r_1 \ldots r_n \in [R_r, R_s]$.

Exemple 3.2.2 *L'ensemble des routes élémentaires de l'exemple de graphe d'architecture de la figure 3.5 est :*

- $r_1 = \{R_a, a_1, C_{a2}, S_{ab}, C_{b1}, a_2, R_b\}$ *(connecte Opr_A et Opr_B),*
- $r_2 = \{R_a, a_1, C_{a1}, S_{ac}, C_{c1}, a_3, R_c\}$ *(connecte Opr_A et Opr_C),*
- $r_3 = \{R_b, a_2, C_{b2}, S_{bd}, C_{d2}, a_4, R_d\}$ *(connecte Opr_B et Opr_D),*
- $r_4 = \{R_c, a_3, C_{c2}, S_{cd}, C_{d1}, a_4, R_d\}$ *(connecte Opr_C et Opr_D).*

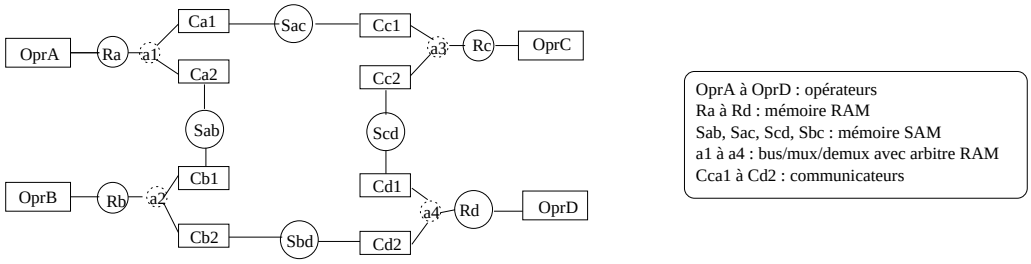


FIG. *3.5:* Routes composées

L'ensemble des routes composées homogènes est :

- $r_5 = r_1 \cup r_3 = \{R_a, a_1, C_{a2}, S_{ab}, C_{b1}, a_2, R_b, a_2, C_{b2}, S_{bd}, C_{d2}, a_4, R_d\}$ *(connecte Opr_A et Opr_D),*
- $r_6 = r_2 \cup r_4 = \{R_a, a_1, C_{a1}, S_{ac}, C_{c1}, a_3, R_c, a_3, C_{c2}, S_{cd}, C_{d1}, a_4, R_d\}$ *(connecte Opr_A et Opr_D)*
- $r_7 = r_3 \cup r_4 = \{R_b, a_2, C_{b2}, S_{bd}, C_{d2}, a_4, R_d, a_4, C_{d1}, S_{cd}, C_{c2}, a_3, R_c\}$ *(connecte Opr_B et Opr_D),*
- $r_8 = r_1 \cup r_2 = \{R_b, a_2, C_{b1}, S_{ab}, C_{a2}, a_1, R_a\, a_1, C_{a1}, S_{ac}, C_{c1}, a_3, R_c\}$ *(connecte Opr_B et Opr_D).*

Les routes r_5 et r_6, ont la même longueur, et permettent par exemple de communiquer des données entre les opérateurs OprA et OprD.

3.2.2.3 Formalisation

Nous appelons $\mathcal{R}_{routage}$ la relation qui, appliquée au graphe d'architecture, construit des chemins (combinaisons de RAM, SAM, communicateurs, Bus/Mux/Demuxet Bus/Mux/Demux/Arbitre :

$$\begin{array}{ccc} G_{ar} & \overset{\mathcal{R}_{routage}}{\longrightarrow} & G'_{ar} \\ (S, H_d) & \overset{\mathcal{R}_{routage}}{\longrightarrow} & (S, R_d) \end{array}$$

3.2.3 Distribution

Étant donné que le nouveau graphe d'architecture comporte de nouveaux types de sommets (S_{RAM}, S_{SAM} et S_{bus}), les deux étapes de la distribution (le partitionnement et la communication) vont ajouter de nouveaux sommets dans le graphe d'algorithme et les associer aux nouveaux sommets du graphe d'architecture.

3.2.3.1 Partitionnement

Il consiste toujours à décomposer le graphe d'algorithme en n sous-graphes disjoints qui seront chacun associé à un sommet opérateur. Par contre, chaque dépendance de données intra-partition (D'_p) doit être associée à l'une des routes iso-opérateur $r \in R_{iso,p}$ de l'opérateur p du graphe d'architecture, on la note donc D'_{p_r}. Les dépendances de données inter-opérateurs doivent être associées à l'une des routes inter-opérateurs $R_{inter_{p,q}}$. Soit D'_{iso_p} l'ensemble des dépendances iso-opérateur d'un opérateur p :

$$D'_{iso_p} = \bigcup_{r \in R_{iso,p}} D'_{p_r}$$

Remarque 22 *Plusieurs dépendances inter-partitions peuvent être distribuées sur la même route r. Nous verrons dans la section Ordonnancement qu'elles seront alors séquentialisées.*

Remarque 23 *Lorsqu'il existe $n > 1$ opérations consommatrices d'une même donnée (hyperarc) et que chaque opération consommatrice est associée à un opérateur différent, l'hyperarc est associé à chacune des n routes élémentaires ou composées permettant de joindre les opérateurs des opérations productrices et consommatrices. Dans le cas de routes composées faites de routes élémentaires communes, nous allons voir qu'il est possible de réutiliser ces routes élémentaires en faisant de la diffusion.*

Le modèle d'architecture présenté ici permet, comme celui de la thèse d'A. Vicard, de décrire les machines hétérogènes. Cependant,

Contrairement aux modèles présenté dans [102], et pour modéliser plus précisément l'implantation sur des architectures hétérogènes, nous ne faisons pas ici l'hypothèse que tous les opérateurs sont capable d'exécuter toutes les opérations du graphe d'algorithme. C'est pourquoi nous introduisons une contrainte, appelée *contrainte de placement*, modélisée par les deux applications suivantes :

Soit l'application λ qui, à chaque opération de calcul, associe l'ensemble des opérateurs sur lesquels il est possible de la distribuer :

$$\begin{array}{rccl} \lambda : & O' & \rightarrow & \mathcal{P}(S_{Opr}) \\ & o'_i & \mapsto & \lambda(o'_i) = \{p_j \in S_{Opr} \ / \ \lambda^{-1}(p_j) = o'_i\} \end{array}$$

Remarque 24 *Chaque opération du graphe d'algorithme doit être exécutable par au moins un opérateur du graphe d'architecture : $\forall o' \in O', \lambda(o') \neq \emptyset$*

Symétriquement, soit λ^{-1} l'application qui, à chaque opérateur, associe l'ensemble des opérations qu'il est capable d'exécuter :

$$\begin{array}{rccl} \lambda^{-1}: & S_{Opr} & \rightarrow & \mathcal{P}(O') \\ & p_j & \mapsto & \lambda^{-1}(p_j) = \{o'_i \in O' \ / \ p_j \subset \lambda(o'_i)\} \end{array}$$

Remarque 25 *Les applications* Π *et* Π^{-1} *présentées précédemment restent valides ici.*

A partir d'un graphe d'algorithme et d'un graphe d'architecture maintenant routé selon deux types de routes (iso-opérateur et inter-opérateurs), le partitionnement permet de construire les graphes partitionnés $G_{partR}(.)$ maintenant définis par :

$$\begin{array}{|cc|} \hline (G_{al}, G'_{ar}) & \stackrel{\mathcal{R}_{Part}}{\longrightarrow} \quad (G_{partR}, G'_{ar})(.) \\ ((O', D'), (S, Rd)) & \stackrel{\mathcal{R}_{Part}}{\longrightarrow} \quad (\bigcup\limits_{p \in S_{cal}} (O'_p, D'_{iso_p}), \bigcup\limits_{r \in R} D'_r) \\ \hline \end{array}$$

Propriété 1 *Cette relation n'a pas ajouté ou retiré d'arcs par rapport à la relation présentée dans la section 3.1.3.1, la démonstration portant sur la conservation de l'ordre partiel reste donc valide ici :*

$$\preceq = (\bigcup_{p \in S_{cal}} \preceq_p) \cup (\bigcup_{r \in R} \preceq_r) \quad avec \quad R = R_{inter} \cup R_{iso}$$

3.2.3.2 Communications

Comme précédemment, la communication d'une dépendance de données inter-opérateurs conduit à ajouter un ensemble de sommets O'' associés aux communicateurs, ainsi que les ensembles d'arcs D'' et $D_P{}^*$.

Soit Π_{com}, l'application qui, à chaque opération de communication, associe l'opérateur de calcul sur lequel elle est distribuée :

$$\begin{array}{rccl} \Pi': & O'' & \rightarrow & S_{com} \\ & o''_i & \mapsto & \Pi_{com}(o''_i) = c_j \end{array}$$

Soit ${\Pi_{com}}'^{-1}$, l'application réciproque de Π_{com}, qui associe à chaque communicateur, l'ensemble des opérations de communication distribuées sur ce communicateur :

$$\begin{array}{rccl} {\Pi_{com}}^{-1}: & S_{com} & \rightarrow & \mathcal{P}(O'') \\ & c_j & \mapsto & \Pi\{com^{-1}(c_j) = \{o''_i \in O'' \ / \ \Pi_{com}(o''_i) = c_j\} \end{array}$$

3.2.3.2.1 Sommets allocation et identité Cependant, comme chaque route renferme maintenant de nouveaux types de sommets (S_{RAM}, S_{SAM} et S_{bus}), la communication d'une dépendance de données associée à une route doit être enrichie. En effet, lorsque des opérations de calcul ou de communication sont en dépendances de données elles doivent se transmettre leurs données en écrivant et en lisant ces données dans une des mémoires qu'elles se partagent. La quantité de mémoire n'étant pas infinie et pour des besoins de génération de code, il est nécessaire d'allouer (réserver un espace) la mémoire dans laquelle les opérateurs et les communicateurs vont lire et écrire ces données. Dans le précédent modèle il n'y avait qu'une seule mémoire

partagée entre l'opérateur et les communicateurs d'un processeur. Dans ce nouveau modèle, les opérateurs et communicateurs peuvent être connectés à plusieurs mémoires, il faut donc indiquer explicitement laquelle de ces mémoires sera allouée pour communiquer chaque dépendance de données.

Pour cela, nous modélisons l'allocation d'une RAM ou d'une SAM par un sommet *allocation* associé à la RAM ou la SAM :

- soit O''_{alloc} l'ensemble des sommets allocation d'un graphe d'algorithme,
- soit α l'application qui, à chaque dépendance de données, associe l'ensemble des sommets allocation issus de la communication (il peut en effet y avoir plusieurs sommets allocation pour une même dépendance quand celle-ci diffuse) :

$$\begin{array}{rccl} \alpha: & \mathcal{P}(D'_{isoP} \cup D'' \cup {D_P}^*) & \rightarrow & O''_{alloc} \\ & d'_i & \mapsto & \alpha(d'_i) = a_j \end{array}$$

- soit α^{-1} l'application qui à chaque sommet allocation associe la dépendance de données correspondante :

$$\begin{array}{rccl} \alpha^{-1}: & O''_{alloc} & \rightarrow & D'_{isoP} \cup D'' \cup {D_P}^* \\ & a_i & \mapsto & \alpha^{-1}(a_i) = \{d'_j / \alpha(d'_j) = a_i\} \end{array}$$

- soit Π_{mem} l'application qui à chaque sommet allocation associe le sommet mémoire sur lequel il est distribué :

$$\begin{array}{rccl} \Pi_{mem}: & O''_{alloc} & \rightarrow & S_{RAM} \cup S_{SAM} \\ & a_i & \mapsto & \Pi_{mem}(a_i) = m_j \end{array}$$

- soit ${\Pi_{mem}}^{-1}$ l'application qui, à chaque sommet mémoire, associe l'ensemble des sommets allocation qui y sont distribués :

$$\begin{array}{rccl} {\Pi_{mem}}^{-1}: & S_{RAM} \cup S_{SAM} & \rightarrow & \mathcal{P}(O''_{alloc}) \\ & m_i & \mapsto & {\Pi_{mem}}^{-1}(m_i) = \{a_j / \Pi_{mem}(a_j) = m_i\} \end{array}$$

Chaque sommet allocation a_i, associé à une mémoire m_j, correspond à l'allocation de la mémoire nécessaire pour transmettre une donnée entre au moins deux opérations connectées par une dépendance de données d_k. A chaque sommet allocation, il est possible d'associer une durée minimale d'allocation. Sa date de début correspond à la première écriture dans l'espace mémoire réservé, cela correspond à la date de début de l'opération productrice de d_k. Sa date de fin correspond à la date de fin de la dernière opération qui fera une lecture de cet espace mémoire, cette opération appartient aux consommateurs de d_k. La date de début et de fin d'un sommet allocation définissent sa durée d'existence (ou durée de vie).

Un sommet allocation n'est pas en relation de dépendance (ordre partiel) avec les deux sommets (calcul ou communication) qui accèdent à la mémoire, il n'est ni prédécesseur (au sens Γ^{-1} défini en 2.1.2.2), ni successeur (Γ) de ces sommets. Cependant, comme il est nécessaire d'associer (relier) le sommet allocation à ces deux sommets connectés par une dépendance de données, nous introduisons un nouveau type de relation d'association entre sommet allocation et sommets opération connectés par une dépendance de données (c'est à dire $\gamma(d_k)$ et $\gamma^{-1}(d_k)$ pour une dépendance de données d_k) . Cette relation est modélisée par un hyperarc non orienté (représenté en pointillés) qui n'est pas une dépendance de données puisqu'il n'introduit pas de relation d'ordre ($\preceq$) entre les sommets. L'ensemble de ces hyperarcs non orientés est noté D''_α, on a $H''_\alpha \subseteq O''_{alloc} \times \mathcal{P}(O' \cup O'')$.

Symétriquement, à chaque sommet Bus/Mux/Demux et Bus/Mux/Demux/Arbitre composant une route, on associe un sommet *identit´e* :

- soit O''_{ident} l'ensemble des opérations identité du graphe,

- soit ι l'application qui à chaque dépendance de données, associe l'ensemble des sommets identité issus de la communication :

$$\begin{array}{rccl} \iota: & \mathcal{P}(D'_{isoP} \cup D'' \cup {D_P}^*) & \rightarrow & O''_{ident} \\ & d'_i & \mapsto & \iota(d'_i) = i_j \end{array}$$

- soit ι^{-1} l'application qui à chaque sommet identité associe la dépendance de données correspondante :

$$\begin{array}{rccl} \iota^{-1}: & O''_{ident} & \rightarrow & D'_{isoP} \cup D'' \cup {D_P}^* \\ & i_i & \mapsto & \iota^{-1}(i_i) = \{d'_j / \iota(d'_j) = i_i\} \end{array}$$

- soit Π_{bus} l'application qui à chaque sommet identité associe le sommet Bus/Mux/Demux ou Bus/Mux/Demux/Arbitre s lequel il est distribué :

$$\begin{array}{rccl} \Pi_{bus}: & O''_{ident} & \rightarrow & S_{bus} \\ & i_i & \mapsto & \Pi_{bus}(i_i) = b_j \end{array}$$

- soit ${\Pi_{bus}}^{-1}$ l'application qui, à chaque sommet Bus/Mux/Demux ou Bus/Mux/Demux/Arbitre associe l'ensemble des sommets identité qui y sont distribués :

$$\begin{array}{rccl} {\Pi_{bus}}^{-1}: & S_{bus} & \rightarrow & \mathcal{P}(O''_{ident}) \\ & b_i & \mapsto & {\Pi_{bus}}^{-1}(b_i) = \{i_j / \Pi_{bus}(i_j) = b_i\} \end{array}$$

Dans le cas des sommets identité, on peut appliquer le même raisonnement que pour les sommets allocation du paragraphe précédent. On note H''_ι l'ensemble des hyperarcs non orientés qui connectent un ou plusieurs sommets identité à des sommets opérations connectées par une dépendance de données, on a $H''_\iota \subseteq \mathcal{P}(O''_{ident}) \times \mathcal{P}(O' \cup O'')$.

3.2.3.2.2 Communication iso-opérateur La communication d'une dépendance de données associée à une route iso-opérateur consiste donc à associer un sommet allocation sur le sommet RAM de cette route, et autant de sommets identité qu'il y a de sommets Bus/Mux/Demux sur cette route (Cf exemple de la figure 3.6 dans laquelle les sommets allocation et identité sont associés aux opérations par un hyperarc représenté en pointillés).

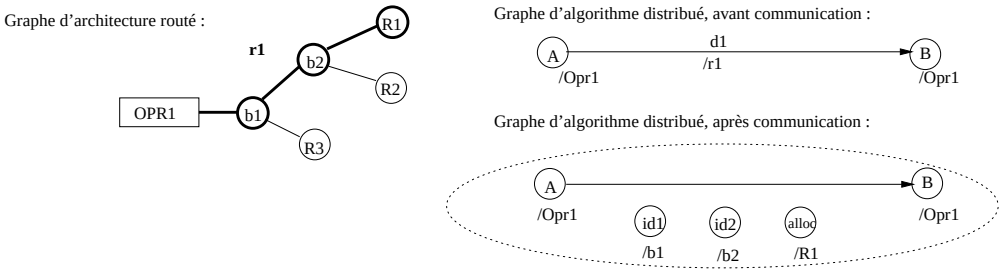


FIG. 3.6: *Communication intra-partition*

Si d est la dépendance à implanter, le sommet allocation est étiqueté par un nom unique obtenu à partir du nom de l'opération productrice de la dépendance implantée (c'est à dire $\gamma^{-1}(d)$) concaténée avec la valeur contenue dans le premier élément du quadruplet associé à d (c'est à dire $Pos_{in}(d)$, position de l'argument dans la liste d'appel de la fonction correspond à l'opération, Cf. Modèle d'algorithme § 2.5.2). Dans la phase d'optimisation et lors de la génération d'exécutif ce nom sera utilisé pour générer des tampons mémoire.

3.2.3.2.3 Communication inter-opérateurs sur route élémentaire Chaque communication inter-opérateurs dépend du type de route qui a été associé à la dépendance de données. Dans ce qui suit nous présentons les différents types de communications qu'il faut effectuer selon les différents types de routes élémentaires ou composées.

- **RAM unique :** les communications sur ce type de route sont les plus simples puisque l'opération productrice écrit dans la RAM les données qu'elle doit communiquer (éventuellement en traversant un ou plusieurs sommets Bus/Mux/Demux). L'opération consommatrice peut ensuite lire ces données dans la RAM partagée en traversant éventuellement des sommets Bus/Mux/Demux (Cf exemple figure 3.7). La dépendance de données est donc transformée en un sous-graphe composé d'un sommet allocation (associé aux RAM de la route) et d'autant de sommets identités qu'il y a de sommets Bus/Mux/Demux sur la route. Ces sommets sont reliés par un hyperarc.

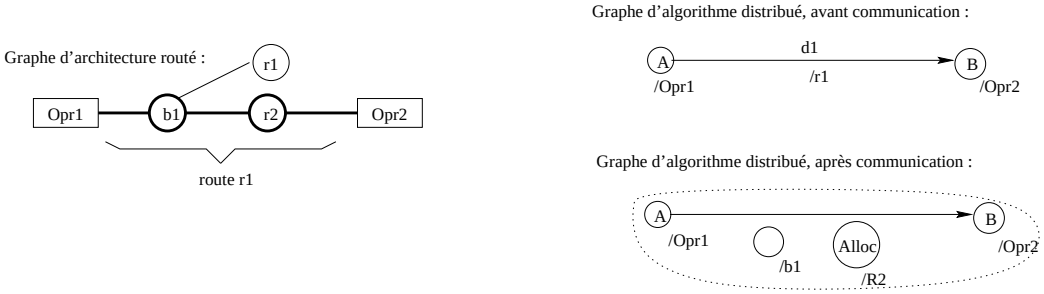


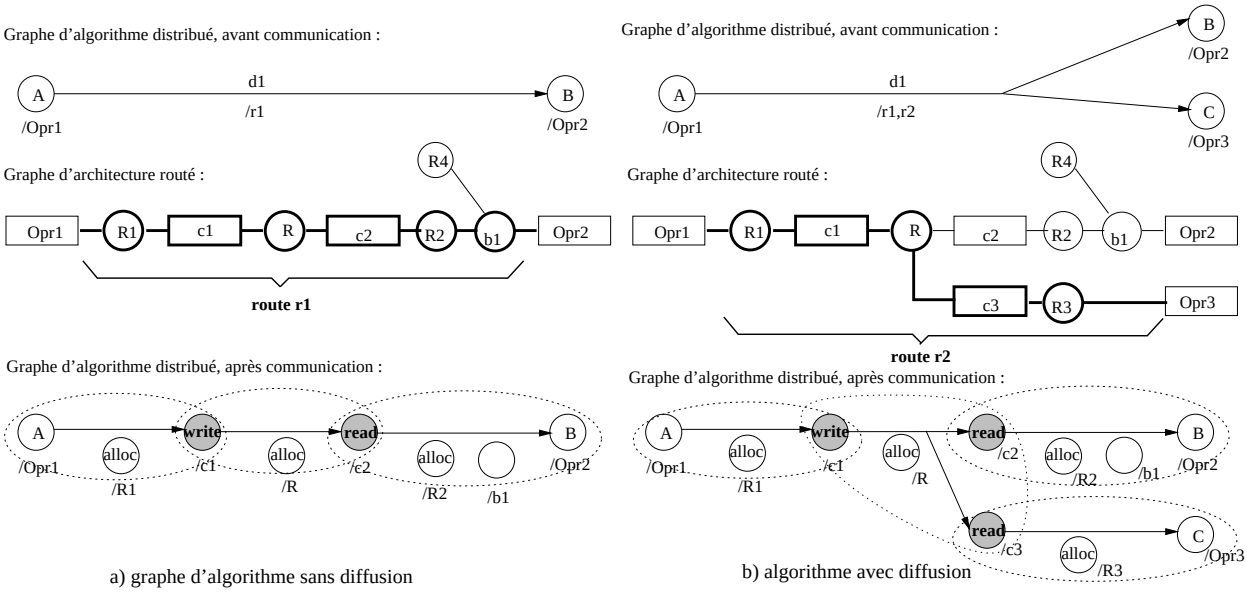
FIG. 3.7: *Graphes d'architecture avec communications par RAM partagé unique*

Remarque 26 *Dans la section "Ordonnancement" nous verrons qu'il faut ajouter un mécanisme pour garantir que l'exécution d'une opération qui fait une lecture dans la RAM commence toujours après la fin de l'opération qui fait l'écriture correspondante.*

- **RAM et communicateurs :** la route entre les opérateurs est composée de deux RAM connectées chacune d'une part à un opérateur, et d'autre part à un communicateur. Ces derniers sont connectés à une troisième RAM qu'ils se partagent (par l'intermédiaire d'éventuels Bus/Mux/Demux et Bus/Mux/Demux/Arbitre si il y a plusieurs communicateurs). La communication d'une dépendance de données associée à ce type de route consiste à remplacer cette dépendance par un sous-graphe composé d'un sommet opération de communication, appelé *write*, associé au premier communicateur de la route, pour transférer les données entre les deux mémoires qui sont connectées à ce communicateur et d'un second sommet opération de communication, appelé *read*, associé au second communicateur, pour lire les données déposées dans la RAM partagée et les écrire dans la RAM connectée à l'opérateur qui exécute l'opération consommatrice de ces données. A chaque dépendance de données $D_p{}^*$ (entre une opération de calcul et une opération de communication) et D'' (entre opérations de communication) est associé un sommet allocation distribué sur la RAM correspondante et autant de sommets identité qu'il y a de Bus/Mux/Demux et Bus/Mux/Demux/Arbitre (Cf exemple figure3.8). Soit O''_{write} (resp. O''_{read} le sous-ensemble des opérations de communication de O'' de type write (resp. read). On a donc $O''_{write} \subset O''$ et $O''_{read} \subset O''$.

Remarque 27 *La mémoire partagée entre les communicateurs étant de type RAM, c'est à dire à accès aléatoire, l'opération* `read` *peut être exécutée bien après l'opération* `write` *correspondante.*

Il est possible d'exécuter plusieurs opérations `write` *avant d'exécuter une des opérations* `read` *(dans la mesure où chaque opération* `write` *écrit dans une zone mémoire différente). L'ordre entre les* `write` *et les* `read` *peut être quelconque, il faut seulement garantir, à l'exécution, qu'un* `read` *ne sera pas exécuté tant que l'exécution du* `write` *correspondant ne sera pas terminée. Dans la section ordonnancement, nous verrons comment ajouter un mécanisme pour garantir que l'exécution d'une opération* `write` *commence toujours après la fin de l'opération de calcul productrice et que symétriquement, l'exécution d'une opération de calcul consommatrice commence toujours après la fin de l'opération* `read` .



a) graphe d'algorithme sans diffusion

b) algorithme avec diffusion

FIG. 3.8: *Communications sur routes de type R_r*

- **SAM point-à-point et communicateurs :** ces routes diffèrent des routes précédentes par la présence d'une SAM partagée entre les communicateurs. Une SAM est une mémoire à accès séquentiel dont la capacité est en général relativement faible vis à vis de la quantité de données qui y transite. Si la quantité de données à communiquer excède sa capacité, ces données doivent être découpées en paquets élémentaires (chacun de taille égale à la capacité de la SAM) écrits tour à tour sur la SAM. La nature séquentielle de la SAM impose que chaque paquet écrit, soit lu avant qu'il soit possible d'en écrire un nouveau. Pour cela, nous avons vu que l'arbitre des SAM fournit toujours une synchronisation matérielle entre l'écriture et la lecture d'une donnée (alors que les RAM ne fournissent aucune synchronisation). Lors d'une communication par SAM, deux cas peuvent donc se présenter selon la quantité de données associée à la dépendance de données :

  1. lorsque la taille n, du bloc de données à communiquer ne dépasse pas la capacité N de la SAM (une FIFO à N place), la communication est réalisée en remplaçant la dépendance de données par un sommet opération de communication émetteur associé au premier communicateur pour écrire toutes les données dans la SAM, et par un sommet opération de communication récepteur associé au second communicateur pour lire les données stockées dans la SAM afin de les copier dans la RAM connectée (cette opération est synchronisée matériellement avec la première

opération de communication). A chaque sommet mémoire (resp. bus) de la route sera associé un sommet allocation (resp. identité).

2. lorsque la taille m, du bloc de données à communiquer dépasse la capacité N de la SAM (Cf exemple b de la figure 3.9), la communication est réalisée en remplaçant la dépendance de données par un sous-graphe resultant de la répétition d'un sous-graphe linéaire correspondant à M/n fois le cas 1) avec de la diffusion. Cela correspond à M/n opérations de communication émetteur (pour transmettre chaque paquet élémentaire de données), associées au premier communicateur et M/n sommets opération de communication récepteur associés au second communicateur. A chaque sommet mémoire (resp. bus) de la route sera associé un sommet allocation (resp. identité).

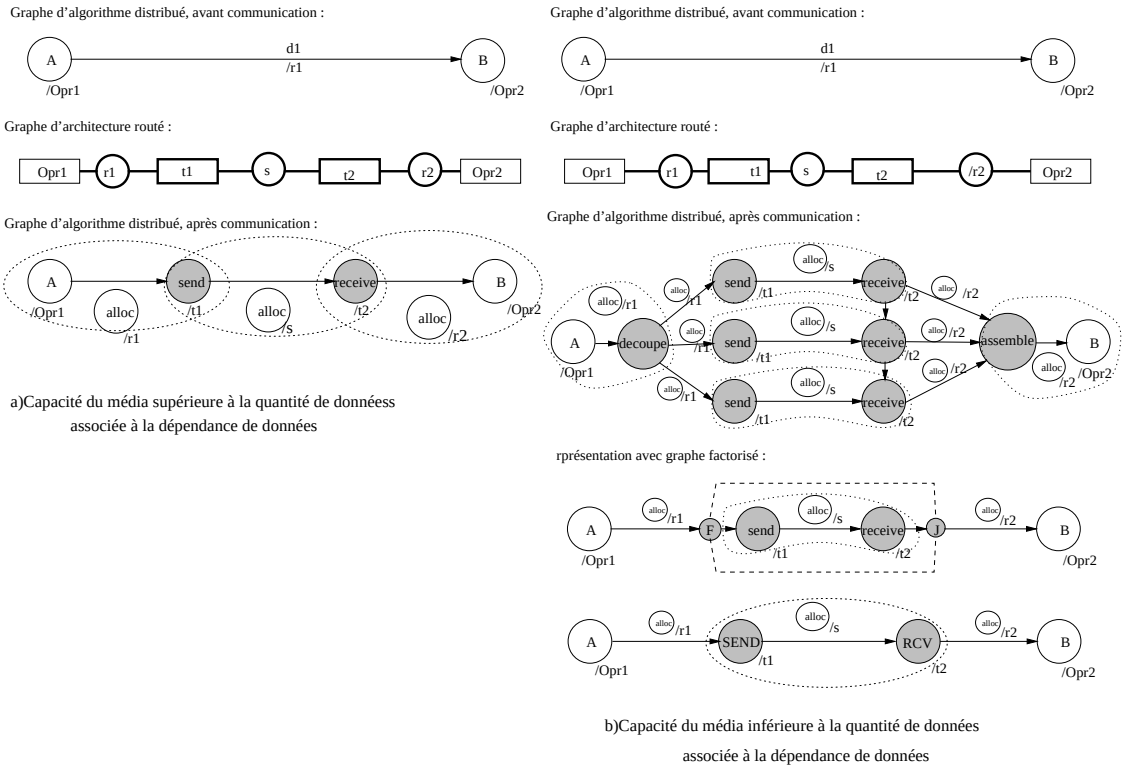


FIG. 3.9: *Communications sur routes de type R_S*

Le premier cas (Cf cas a de la figure 3.9) semble très similaire à la communication par RAM partagée exposée plus haut, cependant dans le cas d'une SAM, aucune autre opération émettrice ne peut être exécutée tant que l'opération réceptrice des données écrites n'a pas été exécutée puisque c'est elle qui vide la SAM. Les sommets opérations de communication émetteurs et récepteurs portent donc des noms différents et seront implantés différemment, nous les nommons `send` et `receive` afin d'éviter toute ambiguïté entre les sommets `write` et `read` qui eux ne sont pas synchronisés (un `write` peut être exécuté sans qu'aucun `read` n'ait lieu ensuite, et ceci sans pour autant bloquer l'accès à la mémoire).

Dans le second cas (quantité des données supérieure à la capacité de la SAM), les données sont découpées en paquets (par une opération *d´ecoupe*) puis chaque `send` lit tour à tour des blocs de données

écrits dans la RAM pour les copier dans une adresse fixe (celle de la SAM). Chaque `receive` lit tour à tour à cette adresse fixe, chaque bloc élémentaire pour les copier à des adresses contiguës dans la RAM afin de reformer le bloc de données initial (opération *assemble*). A l'exécution, nous verrons (§ 4.1.2.4, p. 101), qu'il peut donc y avoir du parallélisme "pipeline" (ou recouvrement) entre l'exécution d'une opération `receive` et d'une opération `send` de la prochaine donnée.

Par souci de simplification, il est possible de modéliser une partie du sous-graphe par un graphe factorisé, tel que présenté sur le cas b de la figure 3.9 suivant le mécanisme expliqué dans le modèle d'algorithme (Cf. § 2.2). Le fork découpe les données et le join les assemble. Par la suite nous considérons que le sommet `fork` (resp. `join`) et les sommets `send` (resp. `receive`) sont encapsulés dans un unique sommet `SEND` (resp. `RECEIVE`). Quelque soit la quantité de données associée à la dépendance de données, celle-ci sera donc remplacée par un sous-graphe composé d'un unique sommet `SEND`, un unique sommet `RECEIVE` ainsi qu'un unique sommet allocation et autant de sommets identité qu'il y a de sommets Bus/Mux/Demux (comme dans le cas a de la figure 3.9, et ce quelque soit la quantité de données à transférer puisque la répétition peut être égale à 1). Ainsi, la représentation temporelle de la communication, montre un recouvrement temporel entre l'exécution d'un SEND et celle d'un RECEIVE. Soit O''_{SEND} (resp. $O''_{RECEIVE}$) le sous-ensemble des opération de communication O'' de type SEND (resp. RECEIVE). On a donc $O''_{SEND} \subseteq O''$ et $O''_{RECEIVE} \subseteq O''$.

- **SAM multipoint et communicateurs :** les communications sur des routes constituées de SAM multipoint se font de manière très similaire aux communications par SAM point-à-point, la différence réside au niveau de la diffusion des données. En effet, il faut distinguer deux types de communication sur les SAM multipoint selon que ces SAM supportent ou non matériellement la diffusion (broadcast) :

  1. **SAM multipoint sans broadcast matériel :** dans le cas où la dépendance de données est un hyperarc associé à un ensemble de routes qui ont toutes une même SAM en commun, et si cette SAM ne supporte pas matériellement la diffusion, il faut répéter le nombre d'opérations de communication `SEND` sur le dernier communicateur commun aux différentes routes. Cela correspond à ajouter autant de sommets `SENDs`, allocations et `RECEIVE` (distribués les communicateurs et la SAM) qu'il y a d'opérations consommatrices distribuées sur des opérateurs destinataires différents (Cf cas a de la figure 3.10),
  2. **SAM multipoint avec broadcast matériel :** la plupart des SAM multipoint supportent la diffusion au niveau matériel. Ainsi, une donnée écrite dans une telle mémoire peut être lue simultanément par plusieurs opérations de communications. Nous allons tirer partie de cette propriété pour optimiser la communication des dépendances de données qui diffusent (hyperarcs). En effet, contrairement au cas précédent, il est alors possible d'exécuter simultanément plusieurs opérations `RECEIVE` correspondant à une même opération `SEND` puisque cette mémoire permet la diffusion vers chaque opération `SEND` distribuée sur les communicateurs connectés à cette SAM et appartenant à l'ensemble des routes associées à la dépendance de données. De plus, nous allons utiliser cette propriété pour synchroniser les communicateurs entre eux (nous en aurons besoin dans la section suivante). Pour cela, lors de la communication d'une dépendance de données distribuée sur une telle route, la dépendance est transformée en un sous-graphe composé d'un sommet `SEND`, d'autant de sommets `RECEIVE` qu'il y a d'opérations consommatrices distribuées sur des opérateurs différents et d'autant d'opérations de synchronisation qu'il y a de communicateurs sur lesquels aucune opération `RECEIVE` n'est distribuée (Cf cas b de la figure 3.10). On appelle *sync* ces opérations de communication n'ayant qu'un rôle de synchronisation (dont le principe sera expliqué en détails dans la prochaine section) car les données qu'elles transférent ne sont jamais utilisées par des opérations de calcul. Soit O''_{sync} ce sous-ensemble des opérations de communication O''.

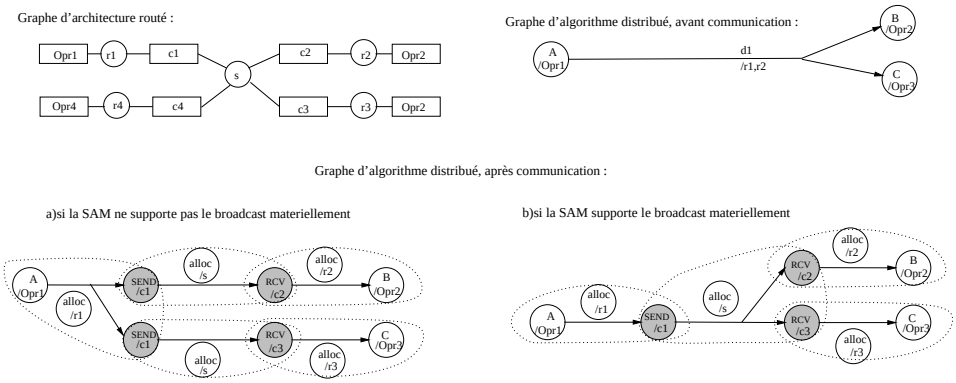


FIG. 3.10: *Communications par SAM multipoint*

3.2.3.2.4 Communication sur route composée Parmi les différentes stratégies de communication de données sur une route composée (le whormole, la commutation de circuit, la communication globale, la commutation de message, Cf. § 1.1.2.3.2), nous avons choisi la communication par commutation de message[89] car les autres sont trop spécifiques à l'architecture sous-jacente ou requièrent un exécutif complexe (qui peut engendrer un surcoût d'exécution et d'espace mémoire code tout en augmentant les risques de deadlock). Dans la communication par commutation de message, les messages avancent sur les routes composées vers leur destination, en passant par les processeurs intermédiaires. A chaque étape, la route élémentaire est aussitôt libérée (cette technique est aussi appelé *store and forward.* Cette méthode suppose implicitement une mémoire suffisante sur chaque mémoire intermédiaire puisqu'il faut stocker le bloc entier. Cela est acceptable ici dans la mesure où, avant distribution, n'importe quel opérateur de l'architecture est susceptible d'exécuter n'importe quelle opération du graphe de l'algorithme (cet ensemble est parfois restreint dans les architectures très hétérogènes, Cf. λ p. 76). De ce fait, chacun des processeurs doit avoir suffisamment d'espace mémoire pour recevoir des données provenant de n'importe quelle opération.

Les communications de dépendance de données sur route composée s'effectuent donc en décomposant chaque route composée en routes élémentaires. La dépendance de données est ainsi communiquée comme expliqué précédemment, sur chaque tronçon de route élémentaire. Nous avons vu (Cf § 3.2.2.2.2) que l'enchaînement des routes élémentaires s'effectue par la mise en commun des RAM aux extrémités de chaque route. De ce fait, l'enchaînement des communications est modélisé au niveau du graphe d'algorithme par la mise en commun des sommets allocation associés aux RAM commune des routes élémentaires.

Si une dépendance de données est associée à un ensemble de routes composées (c'est à dire qu'il y a diffusion) qui ont plusieurs routes élémentaires en commun, la dépendance ne sera communiquée qu'une seule fois sur ces routes communes de façon à optimiser les durées de communication et l'espace mémoire alloué sur chaque mémoire.

3.2.3.3 Allocation mémoire programme

Les instructions qui composent chaque opération exécutée par un opérateur ou un communicateur doivent être stockées dans la mémoire programme connectée à l'opérateur (Cf. § 1.2.2.3 du modèle d'architecture). Nous modélisons l'allocation mémoire programme par des sommets *allocation programme*, O'''_{allocP}, associés aux RAM programmes de l'architecture. Ces dernières faisant nécessairement partie de routes iso-opérateur comportant éventuellement des Bus/Mux/Demux, il faut éventuellement ajouter des sommets identité sur chacun d'eux. Le sommet allocation programme de chaque opération est connecté à l'opération correspondante par un arc non orienté qui, comme précédemment (Cf. § 3.2.3.2.1), n'introduit

pas un ordre partiel puisque ce n'est pas une dépendance de données. L'ensemble de ces arcs non orientés est noté H''_{α_P}, on a $H''_{\alpha_P} \subseteq O''_{allocP} \times O'$.

Soit α_P l'application qui, à chaque opération distribuée sur un opérateur associe un sommet allocation programme :

$$\begin{array}{rccl} \alpha_P : & O' & \rightarrow & O'''_{allocP} \\ & o'_i & \mapsto & \alpha_P(o'_i) = o'''_j \end{array}$$

Soit ${\alpha_P}^{-1}$, l'application réciproque de α_P, qui associe à chaque sommet allocation programme, l'opération correspondante :

$$\begin{array}{rccl} {\alpha_P}^{-1} : & O'''_{allocP} & \rightarrow & O' \\ & o'''_j & \mapsto & {\alpha_P}^{-1}(o'''_j) = \{o'_i \in O' \ / \ \alpha_P(o'_i) = o'''_j\} \end{array}$$

Soit Π_P l'application qui à chaque sommet allocation programme associe le sommet mémoire RAM programme sur lequel il est distribué :

$$\begin{array}{rccl} \Pi_P : & O'''_{allocP} & \rightarrow & S_{SAM} \\ & a_i & \mapsto & \Pi_P(a_i) = m_j \end{array}$$

Soit ${\Pi_P}^{-1}$ l'application qui, à chaque sommet mémoire RAM, associe l'ensemble des sommets allocation programme qui y sont distribués :

$$\begin{array}{rccl} {\Pi_P}^{-1} : & S_{RAM} & \rightarrow & \mathcal{P}(O'''_{allocP}) \\ & m_i & \mapsto & {\Pi_P}^{-1}(m_i) = \{a_j / \Pi_P(a_j) = m_i\} \end{array}$$

Comme la spécification de l'algorithme d'une application est implicitement factorisée infiniment, chaque opération associée à un processeur est exécutée un nombre infini de fois. L'espace mémoire programme réservé pour le stockage des instructions est donc utilisé à chaque itération de l'application pour exécuter l'opération correspondante. Comme cet espace mémoire ne peut être ré-alloué au cours de l'exécution d'un algorithme, la durée de vie d'un sommet allocation programme est donc égale à celle de l'application.

3.2.3.4 Allocation mémoire données locales

La plupart des opérations exécutées par chaque opérateur utilisent des données locales pour leurs calculs, comme nous l'avons vu dans le paragraphe 1.2.2.3) du modèle d'architecture. L'espace mémoire correspondant à l'ensemble des données locales à chaque opération est modélisé par un sommet *allocation donn´ees locales* O'''_{allocP} associé à l'une des RAM connectée à l'opérateur qui exécute l'opération. Cette RAM faisant nécessairement partie d'une route iso-opérateur comportant éventuellement des Bus/Mux/Demux, il faut éventuellement ajouter des sommets identité sur chacun d'eux. Le sommet allocation données locales de chaque opération est connecté à l'opération correspondante par un arc non orienté qui, comme précédemment (Cf. § 3.2.3.2.1), n'introduit pas un ordre partiel puisque ce n'est pas une dépendance de données. L'ensemble de ces arcs non orientés est noté H''_{α_D}, on a $H''_{\alpha_D} \subseteq O''_{allocD} \times O'$.

Exemple 3.2.3 *La figure suivante (3.11) présente les différents types de sommets allocation que nous venons de définir, à travers un exemple d'algorithme composé de deux opérations distribuées sur un même opérateur connecté à une mémoire programme et une mémoire données.*

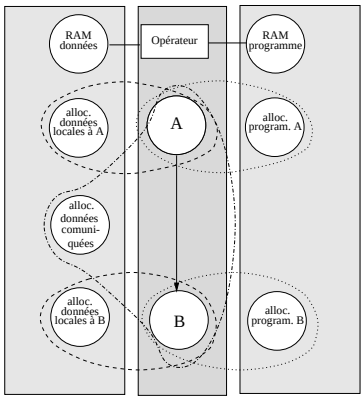


FIG. 3.11: *Les différents types de sommets allocation*

Soit α_D l'application qui, à chaque opération distribuée sur un opérateur associe un sommet allocation données locales :

$$\begin{array}{rccl} \alpha_D : & O' & \rightarrow & O'''_{allocD} \\ & o'_i & \mapsto & \alpha_D(o'_i) = o'''_j \end{array}$$

Soit ${\alpha_D}^{-1}$, l'application réciproque de α_D, qui associe à chaque sommet allocation données locales, l'opération correspondante :

$$\begin{array}{rccl} {\alpha_D}^{-1} : & O'''_{allocD} & \rightarrow & O' \\ & o'''_j & \mapsto & {\alpha_D}^{-1}(o'''_j) = \{o'_i \in O' \ / \ \alpha_D(o'_i) = o'''_j\} \end{array}$$

Soit Π_D l'application qui à chaque sommet allocation données locale associe le sommet mémoire RAM programme sur lequel il est distribué :

$$\begin{array}{rccl} \Pi_D : & O'''_{allocD} & \rightarrow & S_{SAM} \\ & a_i & \mapsto & \Pi_D(a_i) = m_j \end{array}$$

Soit ${\Pi_D}^{-1}$ l'application qui, à chaque sommet mémoire RAM, associe l'ensemble des sommets allocation données locales qui y sont distribués :

$$\begin{array}{rccl} {\Pi_D}^{-1} : & S_{RAM} & \rightarrow & \mathcal{P}(O'''_{alloc}) \\ & m_i & \mapsto & {\Pi_D}^{-1}(m_i) = \{a_j / \Pi_D(a_j) = m_i\} \end{array}$$

3.2.3.5 Représentation graphique des hyperarcs

Comme dans la représentation de graphe d'algorithme, les hyperarcs de dépendances de données sont modélisés par des flèches, avec plusieurs extrémités dans le cas de diffusion.

Chaque hyperarc (de H_α) qui relie un sommet allocation à un ensemble de sommets opération en dépendance de données, est représenté par un trait en pointillés reliant le sommet allocation et la dépendance de données correspondante. Cela permet d'alléger sensiblement la représentation des graphes d'implantation (Cf. exemple ci dessous, figure 3.12). Les sommets identité sont reliés de la même façon au dépendances de données.

Les arcs non orientés qui connectent les sommets allocation programme aux opérations (arcs de H_{α_P}) ainsi que les arcs non orientés qui connectent les sommets allocation données locales aux opérations (arcs de H_{α_D}), sont représentés par des traits en pointillés entre ces sommets.

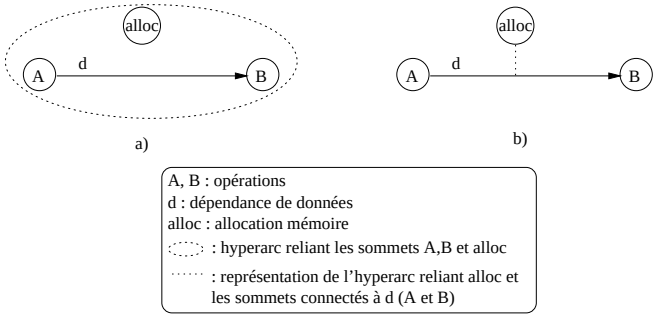


FIG. 3.12: *Représentation des hyperarcs de H_α*

Exemple 3.2.4 *La figure 3.13 présente un exemple de graphe d'algorithme distribué sur un graphe d'architecture composé de trois opérateurs. Les sommets grisés représentent les sommets opérations de communication. Les dépendances d_2 et d_3 ont été distribuées sur la seule route iso-opérateur de $Opr1$. Les dépendances d_1, d_4 et d_6 ont été distribuées sur les routes élémentaires connectant respectivement $Opr1$ et $Opr2$, $Opr2$ et $Opr3$, $Opr1$ et $Opr3$. Enfin la dépendance d_5 a été distribuée sur la seule route composée connectant $Opr1$ et $Opr3$.*

3.2.3.6 Cas des dépendances de conditionnement

Lorsqu'une opération est conditionnée, et que cette opération produit des données pour des opérations exécutées par d'autres opérateurs, il est nécessaire de conditionner les couples d'opérations de communications qui transfèrent ces données.

Du coté émetteur, la dépendance de conditionnement diffuse vers l'opération de calcul productrice et toutes les opérations de communication émettrice (`send` et `write`). Il faut aussi communiquer la dépendance de conditionnement, comme une dépendance de données quelconque, vers tous les communicateurs qui exécutent des opérations consommatrices. Ainsi, du côté récepteur, la dépendance de conditionnement est connectée aux opérations réceptrices (`receive` et `read`).

Exemple 3.2.5 *La figure 3.14 a) présente un graphe d'algorithme dans lequel la sortie de l'opération A conditionne l'exécution de l'opération B qui elle même produit des données (dépendance d_{BC}) à destination de l'opération C. Si A et B sont exécutées par l'opérateur $Opr1$ et que C est exécutée par l'opérateur $Opr2$, il est nécessaire de communiquer d_{BC} en ajoutant les opérations de communication $send_{BC}$ et $receive_{BC}$. Ces deux opérations sont aussi conditionnées par A, et il est donc nécessaire de communiquer la donnée qu'elle produit pour conditionner $receive_{BC}$.*

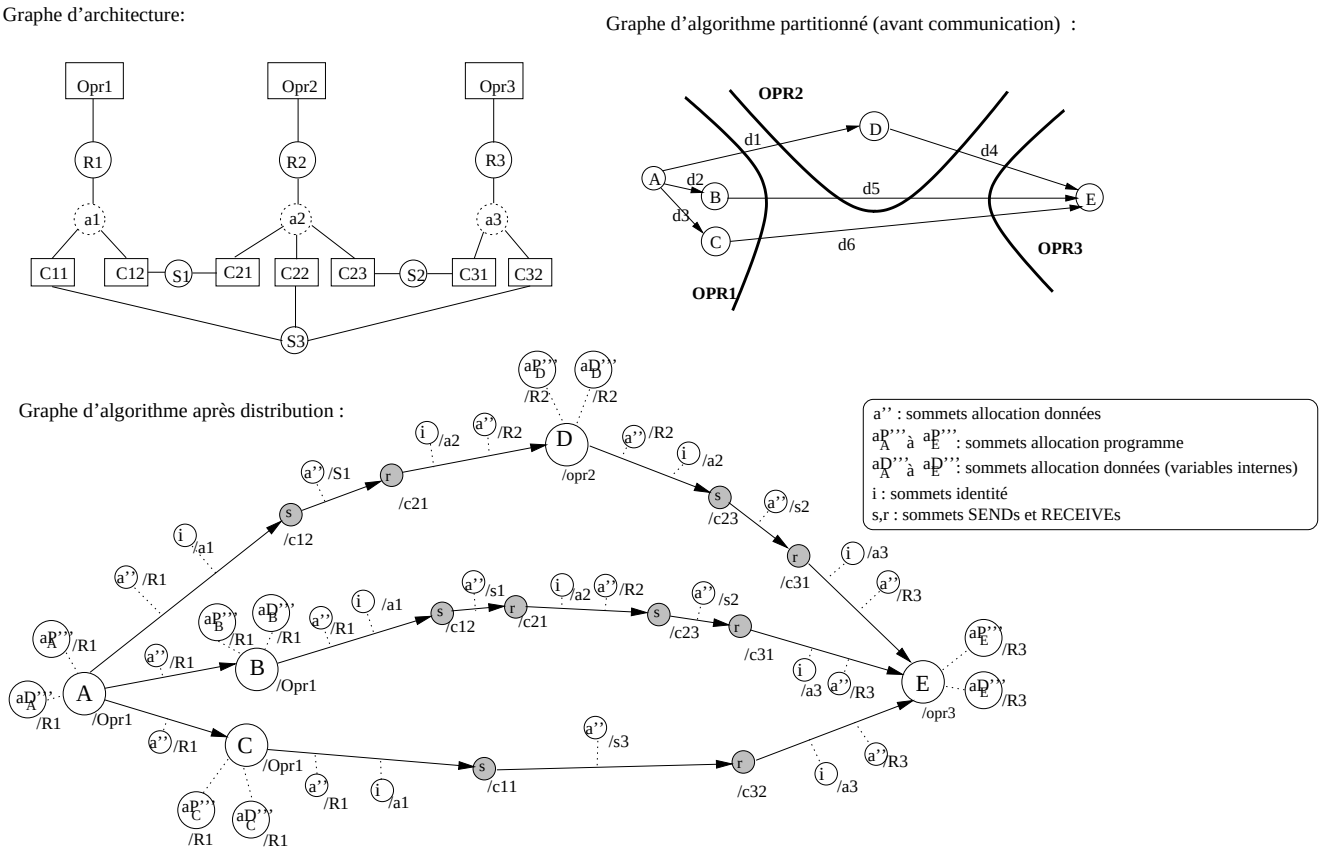


FIG. 3.13: *Exemple d'algorithme distribué*

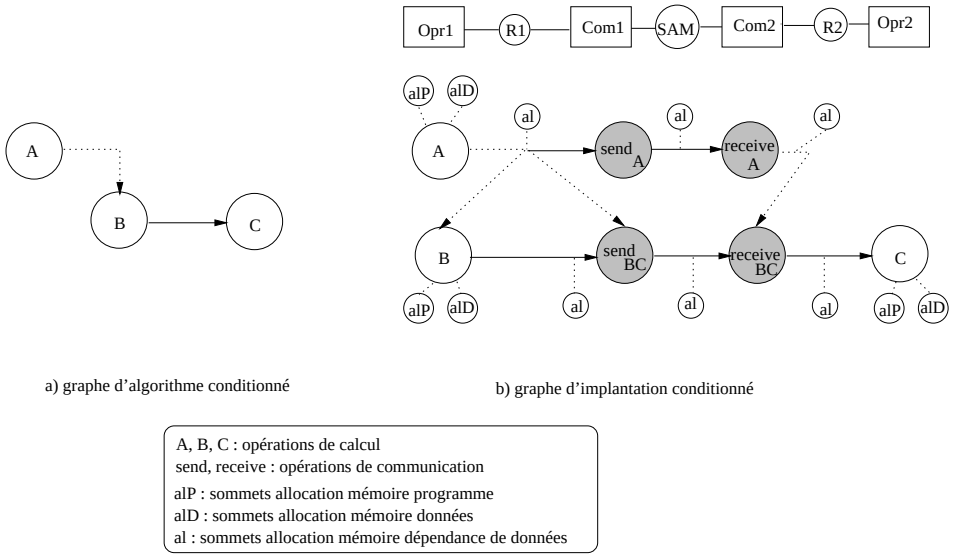


FIG. 3.14: *Dépendance de conditionnement*

3.2.3.7 Formalisation

Pour modéliser la communication sur le nouveau modèle d'architecture nous avons enrichi le modèle de communication par des sommets allocation (O''_{alloc} et O'''_{alloc}) et identité (O''_{ident}) associés respectivement aux sommets mémoires (S_{RAM} et S_{SAM}) et Bus/Mux/Demux, Bus/Mux/Demux/Arbitre S_{bus}). A partir d'un graphe partitionné, la relation appelée communication, notée $\mathcal{R}_{com}$ conduit à un ensemble de graphes $G_{comR}(.)$:

$$
\begin{array}{rcl}
(G_{parR}, G'_{ar}) & \xrightarrow{\mathcal{R}_{com}} & (G_{comR}, G'_{ar})(.) \\
(\bigcup_{p\in S_{cal}} (O'_p, D'_{iso_p}), \bigcup_{r\in R} D'_r) & \xrightarrow{\mathcal{R}_{com}} & ((\bigcup_{p\in S_{cal}} O'_p) \cup (\bigcup_{c\in S_{com}} O''_c) \cup (\bigcup_{s\in S_{mem}} O''_{alloc_s}) \cup (\bigcup_{s\in S_{mem}} O'''_{allocP_s}) \\
 & & \cup(\bigcup_{s\in S_{mem}} O'''_{allocD_s}) \cup (\bigcup_{b\in S_{bus}} O''_{ident_b})), \\
 & & D'_P \cup D''_C \cup D^*_P \cup H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D}) \\
(\bigcup_{p\in S_{cal}} (O'_p, D'_{iso_p}), \bigcup_{r\in R} D'_r) & \xrightarrow{\mathcal{R}_{com}} & (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}}, \\
 & & D'_P \cup D''_C \cup D^*_P \cup H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D})
\end{array}
$$

Avec, comme dans le précédent modèle :

- O'' l'ensemble des opérations de communications ajoutées au graphe de l'architecture, on a maintenant $O'' = O''_{SEND} \cup O''_{RECEIVE} \cup O''_{sync} \cup O''_{read} \cup O''_{write}$,
- D''_C l'ensemble des arcs entre opérations de communication distribuées sur des opérateurs de communication appartenant à des processeurs différents, il induit un ordre partiel $\preceq_c$,
- D^*_{PC}, l'ensemble des arcs entre opérations de calcul et de communication et entre opérations de communication, distribuées sur les même processeurs. D^*_{PC} induit un ordre partiel $\preceq_{pc}$.

Propriété 2 *Comme les hyperarcs H''_{α}, H''_{α_P} et H''_{α_D} ne sont pas des dépendances de données et que nous n'avons pas ajouté de nouvelles dépendances de données dans ce modèle, la conservation de l'ordre partiel démontré dans le paragraphe 3.1.4 reste valide ici :*

$$\preceq \subseteq (\preceq_P \cup \preceq_c \cup \preceq_{pc})$$

Remarque 28 *L'ensemble O'' des opérations de communication a été enrichi des opérations* `read` *,* `write` *et* `sync` *, les opérations de communication de l'ancien modèle correspondent au sommets* `SEND` *,* `RECEIVE` *.*

La distribution (partitionnement et communications) décrite dans ce chapitre permet d'optimiser l'allocation des mémoires puisqu'elle a été modélisée finement au niveau des communications intra-partition sur les routes iso-opérateur, ainsi qu'au niveau des communications inter-partitions utilisant des RAM.

Ce nouveau modèle permet aussi d'optimiser les communications inter-processeurs dans le cas de la diffusion de données, en minimisant le nombre d'opérations de communication (par réutilisation de communications) ce qui a pour double conséquence de minimiser les durées de transferts de données entre opérateurs, mais aussi de minimiser l'espace mémoire alloué sur chaque mémoire de chaque route.

3.2.4 Ordonnancement

Comme dans le précédent modèle (Cf. § 3.1.4), il est maintenant nécessaire de construire un ordre total sur chacun des éléments de partition (alloués aux processeurs) obtenus lors de la distribution. Pour les opérateurs, il est réalisé par l'ajout d'arcs de précédence D'''_{opr} entre les opérations associées à chaque opérateur. Pour les communicateurs, il est réalisé par l'ajout d'arcs de précédence D'''_{com} entre opérations de communications d'un même communicateur. Le nouveau modèle d'architecture, auquel ont été ajoutés les sommets RAM, SAM point-à-point, SAM multipoint (avec support ou non du broadcast), Bus/Mux/Demux et Bus/Mux/Demux/Arbitre nous conduit à enrichir la relation de distribution par l'ajout de nouveaux sommets : allocation, identité, `write` , `read` et `sync` (`SEND` et `RECEIVE` correspondent aux opérations de communication de l'ancien modèle). Dans cette section nous allons enrichir la relation d'ordonnancement en fonction des modifications présentées précédemment.

3.2.4.1 Sommets allocation et identité

L'ajout des sommets allocation et identité ne modifie pas la relation d'ordonnancement présentée dans le paragraphe 3.1.4 car ces sommets ne sont pas associés à des séquenceurs. Nous verrons, lors du chapitre consacré à l'optimisation de la mémoire, qu'il est possible d'étudier les relations d'ordre entre ces sommets pour effectuer de la ré-allocation mémoire, mais cela ne correspond pas à un ordonnancement comme nous le décrivons ici.

3.2.4.2 Contraintes d'ordonnancement imposées par les RAM

Les sommets `read` et `write` sont spécifiques aux communications par mémoires RAM partagées. A l'inverse des mémoires SAM que nous allons étudier en détail dans la section suivante, ces mémoires RAM n'imposent aucun ordre entre les opérations de lectures et d'écritures de leur contenu. Le renforcement de l'ordre partiel en un ordre total (par des dépendances D'''_{com}) entre les opérations logiquement indépendante d'un communicateur connecté uniquement à des RAM, peut donc être effectué librement.

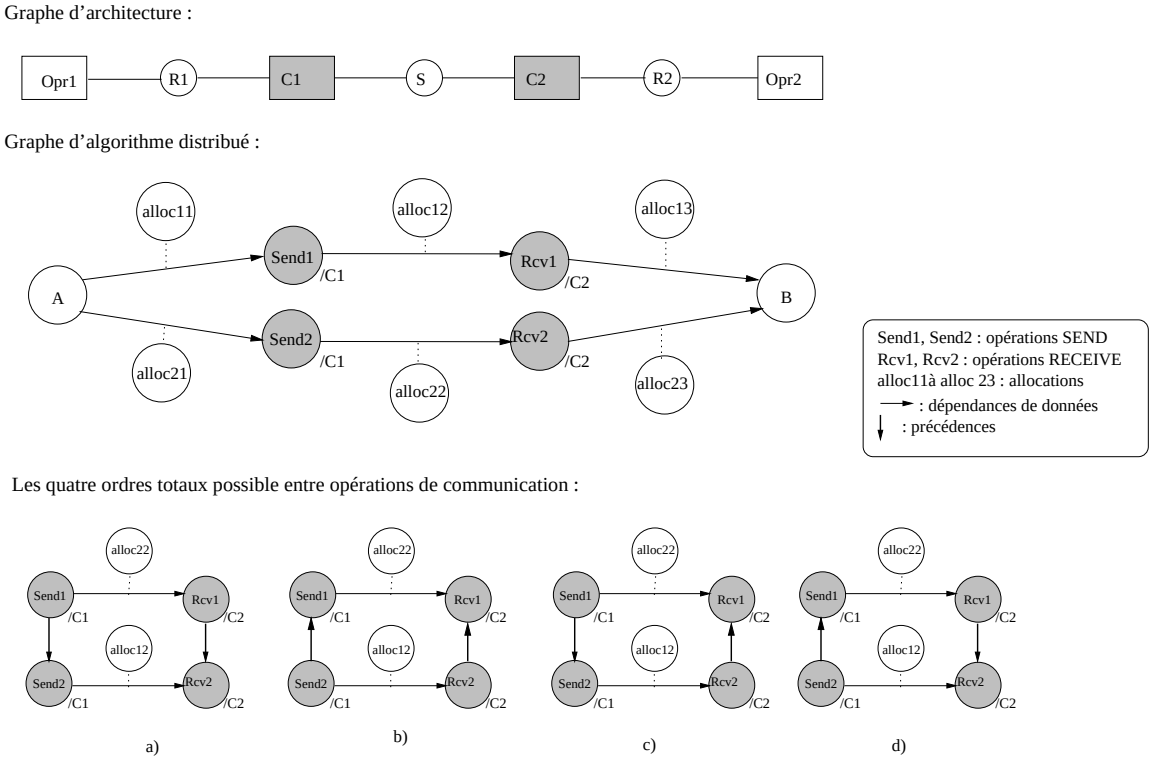


FIG. 3.15: *Contraintes d'ordonnancement intra communicateur imposées par les SAM*

3.2.4.3 Contraintes supplémentaires d'ordonnancement imposées par les SAM

La nature séquentielle des SAM nécessite de construire un ordonnancement sur les opérations de communication de la partition allouée à un communicateur afin de garantir une exécution sans interblocage. Le modèle d'architecture modélisant maintenant précisément les SAM point à point, les SAM multipoint sans support matériel du broadcast et les SAM multipoint supportant le broadcast matériellement, il est nécessaire d'étudier les conséquences de ces différents types de SAM sur l'ordonnancement des opérations de communication de type `SEND` et `RECEIVE` .

3.2.4.3.1 SAM point à point Quand une SAM est partagée par des communicateurs, il est indispensable d'étudier les ordres totaux associés à chaque communicateur car il y a un risque d'interblocage. En effet, étant donné la nature séquentielle d'une SAM, toute donnée écrite dans la SAM (par une opération `SEND` associée à un communicateur connecté à cette SAM) doit être lue (par une opération `RECEIVE` associée à un autre communicateur connecté à la SAM) avant qu'une autre donnée puisse y être écrite.

Ainsi, sur l'exemple de la figure 3.15 où les opérations `rcv1` et `rcv2` ne sont pas logiquement dépendantes, il existe quatre ordres totaux possibles (figures 3.15 a, b, c et d) dont deux (figures 3.15 a et b) sont valides. Les autres (figures 3.15 c et d) génèrent un interblocage. En effet dans les cas a et b les données sont bien lues dans l'ordre où elles ont été écrites. Dans le cas c, l'opération de lecture `rcv1` doit attendre la fin de l'exécution de `rcv2` pour pouvoir lire les données écrites par `send1` , or les données qu'attend `rcv2` n'ont pas pu être écrites par `send2` car les données déposées par `send1` sont toujours bloquées dans la SAM : en effet, `cv1` ne peut être exécuté. La SAM est bloquée, cette situation est appelée *deadlock*. Le cas d amène à la même situation de blocage.

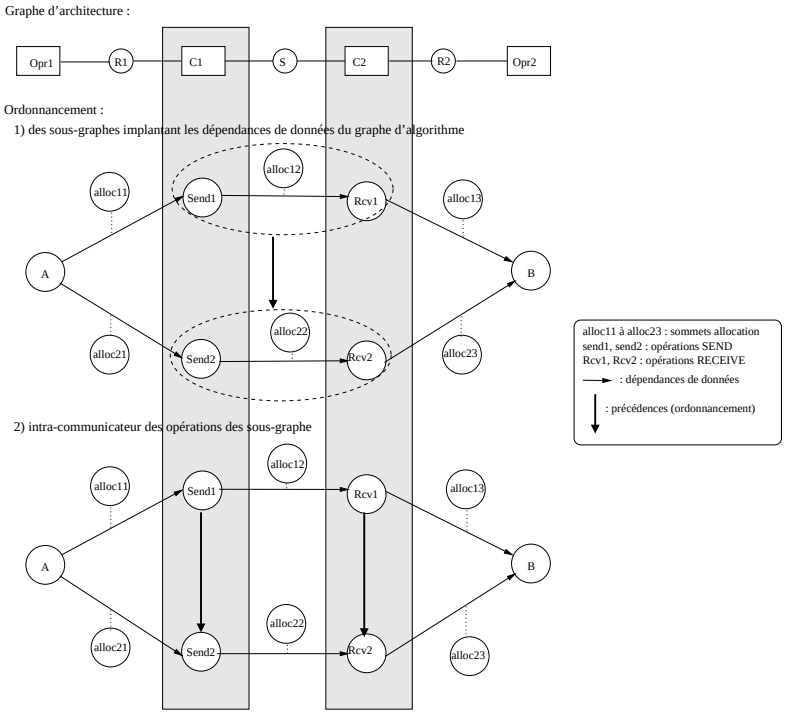


FIG. 3.16: *Ajout des précédences intra-communicateur*

Pour garantir une exécution sans interblocage il est nécessaire de renforcer l'ordre partiel au sein de chaque partition associée à chaque communicateur connecté à une SAM. Cela consiste à ajouter des arcs de précédence intra-communicateur (D'''_{com}), qui ne sont pas des dépendances de données) entre les opérations de communication. La réalisation de ce type de précédence correspond à la séquentialisation des opérations exécutées par le même communicateur.

L'ajout des précédences peut s'effectuer en deux étapes. Tout d'abord il faut considérer chaque sous-ensemble de sommets `SEND` et `RECEIVE` implantant une même dépendance de données comme un sous-graphe (Cf. étape 1 de la figure 3.16). Ensuite il faut établir un ordre partiel entre ces sous-graphes en ajoutant des arcs de précédence entre les opérations de communications de chaque sous graphe distribuées sur les même communicateurs (Cf. étape 2 de la figure 3.16).

Ce choix d'ordonnancement valide sera effectué par les heuristiques d'optimisation (de même que l'ordonnancement des opérations distribuées sur les sommets opérateurs).

Propriété 3 *La consistance du graphe initial de l'algorithme est garantie si, lors de l'implantation, l'ajout des précédences respecte toujours l'ordre partiel des dépendances de données du graphe d'algorithme (Cf. [102]).*

3.2.4.3.2 SAM multipoint sans support du broadcast Dans les paragraphes qui suivent, nous traitons explicitement de l'ordonnancement dans le cas des communications par SAM multipoint, celles-ci n'étaient effectivement pas traitées dans [102]. Dans le cas des SAM multipoint, l'ajout de précédences intra-communicateur ne suffit pas à garantir une exécution sans interblocage. En effet, considérons le cas où deux opérations de communication de type `SEND` (`Send1` et `Send2` sur la figure 3.17) sont exécutées par deux communicateurs différents (`C1 et C2`), et qu'un troisième communicateur (`C3`) exécute

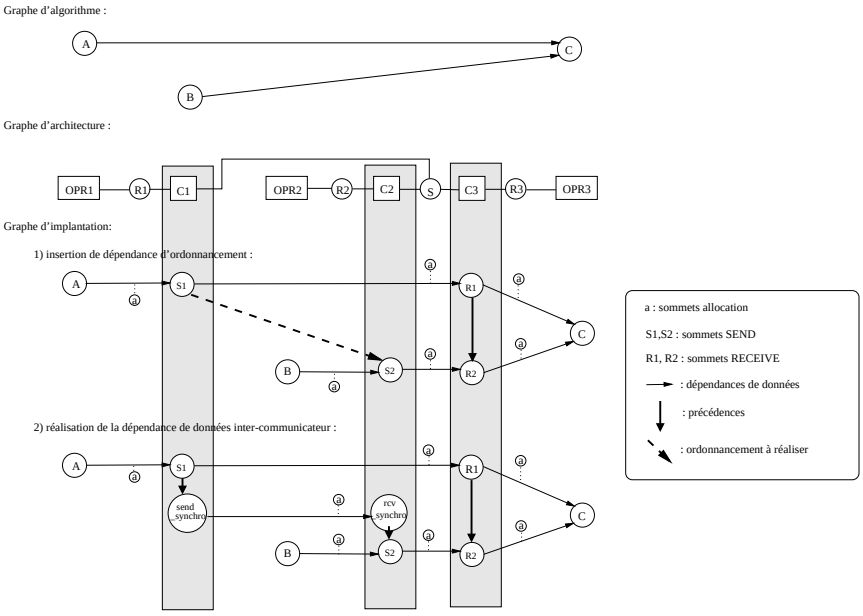


FIG. 3.17: *Ordonnancement inter-communicateurs pour SAM multipoint sans broadcast*

les deux opérations de communication (`Rcv1` et `Rcv2`) correspondantes. Si `Send2` est exécuté avant `Send1` mais que `Rcv1` est exécuté avant `Rcv2`, alors `Rcv2` va attendre des données qui n'arriveront jamais puisque `Send1` doit attendre que les données écrites par `Send2` soient lues pour pouvoir écrire à son tour.

Pour empêcher une telle situation d'interblocage, il est nécessaire d'ajouter des précédences entre les opérations de communication exécutées par le communicateur récepteur, mais aussi entre les opérations de communication exécutées par des communicateurs différents. Étant donné que ces dépendances sont inter-communicateurs, si on choisit par exemple d'exécuter `Rcv1` avant `Rcv2` en ajoutant une précédence intra-communicateur entre ces deux opérations, il faut s'assurer que le `send1` sera exécuté avant `Send2` en ajoutant une précédence inter-communicateurs entre ces deux opérations de communication.

La réalisation de ces précédences inter-communicateurs ne correspond pas à une simple mise en séquence des opérations de communication puisqu'elles connectent des opérations exécutées par des communicateurs différents. Elles peuvent cependant être réalisées en les remplaçant par des couples de sommets `SEND`/`RECEIVE`. La valeur des données transférées par ces opérations n'est alors pas utilisée, ce couple est uniquement utilisé pour synchroniser les communicateurs. Nous ajouterons le qualificatif "synchro" à ces opérations de communications particulières. Le `Send_synchro` doit être ajouté après le premier `SEND` utile (celui qui implante une dépendance de données), et le `Rcv_synchro` correspondant doit précéder l'exécution du second `SEND` utile.

Sur l'exemple de la figure 3.17, cela consiste donc à ajouter un sommet `Send_synchro` après l'exécution de `Send1` et à ajouter le sommet `Rcv_synchro` correspondant, avant `Send2`. Ainsi, `Send2` ne peut commencer tant que `Send1` n'a pas été exécuté.

Remarque 29 *Ce mécanisme peut être comparé au passage d'un jeton entre les différents communicateurs. La réception du jeton correspond au sommet* `Rcv_synchro`. *La possession du jeton permet d'écrire dans la SAM.*

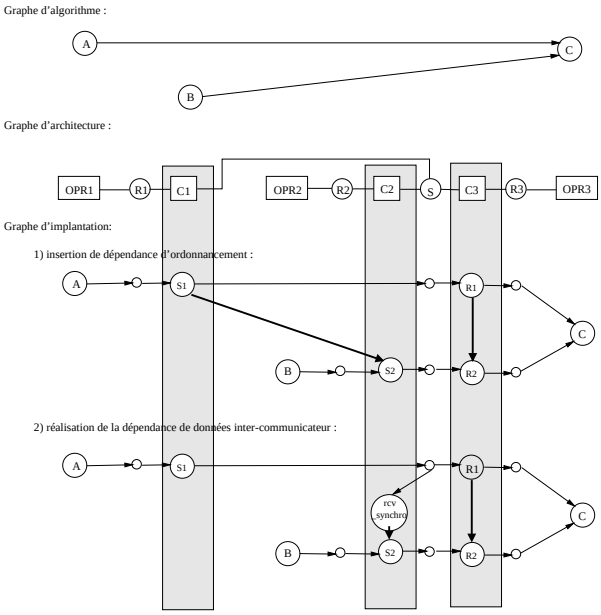


FIG. 3.18: *Ordonnancement inter-communicateurs pour SAM multipoint avec broadcast*

3.2.4.3.3 SAM multipoint avec support du broadcast Les mêmes risques d'interblocage existent quand la SAM supporte matériellement la diffusion (cas du bus CAN[106] par exemple), ce qui nécessite d'ajouter les mêmes précédences intra et inter-communicateurs. Cependant, comme ces SAM supportent matériellement la diffusion, il est possible d'optimiser sensiblement la réalisation des précédences inter-communicateurs. En effet, tirant parti du fait que les données écrites dans la SAM peuvent être simultanément lues par tous les communicateurs, l'ajout d'un sommet `send_synchro` à destination d'un communicateur spécifique est devenue inutile, seule l'opération `rcv_sychro` doit être ajoutée sur chaque communicateur non destinataire d'un `SEND`. Chaque dépendance de données distribuée sur une route composée d'une SAM multipoint avec support matériel du broadcast est donc transformée en un sous-graphe composé d'un sommet `SEND`, d'autant de sommets `RECEIVE` qu'il y a d'opérations consommatrices distribuées sur des opérateurs différents et d'autant d'opérations de synchronisation qu'il y a de communicateurs sur lesquels aucune opération `RECEIVE` n'est distribuée (Cf. figure 3.18).

3.2.4.4 Formalisation

Comme dans la section 3.1.4, l'ordonnancement a ici été réalisé par l'ajout de précédences qui renforcent l'ordre partiel du graphe distribué :

- on renforce donc l'ordre partiel D'_p ($\preceq_p$) de chaque partition associée à un opérateur $p \in S_{opr}$, en un ordre total $\bar{D}'_p$ ($\prec_p$) à l'aide de précédences D'''_p entre les opérations O' du graphe d'algorithme ($\bar{D}'_p = D'_p \cup D'''_p$).

- de même on renforce l'ordre partiel D''_c ($\preceq_c$) de chaque communicateur $c \in S_{com}$ en un ordre total $\bar{D}''_c$ ($\prec_c$) à l'aide de dépendances D'''_c entre les opérations de communication O'' ($\bar{D}_c = D''_c \cup D'''_c$)

- l'ordre partiel $\preceq_{pc}$ donné par les dépendances de données $D_{pc}{}^*$ est inchangé.

Après ordonnancement, l'ensemble des dépendances du graphe d'algorithme distribué et ordonnancé (nous l'appellerons graphe d'implantation) est défini par :

$$D = (\bigcup_{p \in S_{opr}} \bar{D}'_p) \cup (\bigcup_{c \in S_{com}} \bar{D}''_c) \cup (\bigcup_{p \in S_{opr}, c \in S_{com}} D^*_{pc}) = \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}$$

Comme aucun arc n'a été ajouté entre les nouveaux sommets allocation et identité du graphe distribué, l'ordre partiel du graphe d'algorithme peut toujours être décrit par :

$$\trianglelefteq = (\bigcup_{p \in s_{opr}} \prec_p) \cup (\bigcup_{c \in S_{com}} \prec_c) \cup (\bigcup_{p \in S_{opr}, c \in S_{com}} \preceq_{pc})$$

A partir d'un graphe G_{comR} issu de la communication, la relation d'ordonnancement permet de construire l'ensemble des graphes ordonnancés $G_{ordo}(.)$:

$$\begin{array}{rcl}
(G_{comR}, G'_{ar}) & \xrightarrow{\mathcal{R}_{Ordo}} & (G_{ordoR}, G'_{ar})(.) \\
((\bigcup_{p \in S_{cal}} O'_p) \cup (\bigcup_{c \in S_{com}} O''_c) \cup (\bigcup_{s \in S_{mem}} O''_{alloc_s}) & \xrightarrow{\mathcal{R}_{Ordo}} & ((\bigcup_{p \in S_{cal}} O'_p) \cup (\bigcup_{c \in S_{com}} O''_c) \cup (\bigcup_{s \in S_{mem}} O''_{alloc_s}) \\
\cup (\bigcup_{s \in S_{mem}} O'''_{allocP_s}) \cup (\bigcup_{s \in S_{mem}} O'''_{allocD_s}) \cup & & \cup (\bigcup_{s \in S_{mem}} O'''_{allocP_s}) \cup (\bigcup_{s \in S_{mem}} O'''_{allocD_s}) \cup \\
(\bigcup_{b \in S_{bus}} O''_{ident_b}), H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} & & (\bigcup_{b \in S_{bus}} O''_{ident_b}), H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} \\
\cup D'_P \cup D''_C \cup D^*_{PC}) & & \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}) \\
(O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} & \xrightarrow{\mathcal{R}_{Ordo}} & (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \\
\cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}}, & & \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}}, \\
H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup D'_P \cup D''_C \cup D^*_{PC}) & & H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC})
\end{array}$$

Exemple 3.2.6 *La figure 3.19 présente le graphe d'implantation de l'exemple 3.13, après la phase d'ordonnancement. Les arcs en gras représentent les précédences ajoutées pour garantir une exécution sans interblocage, ce qui conduit à une implantation valide.*

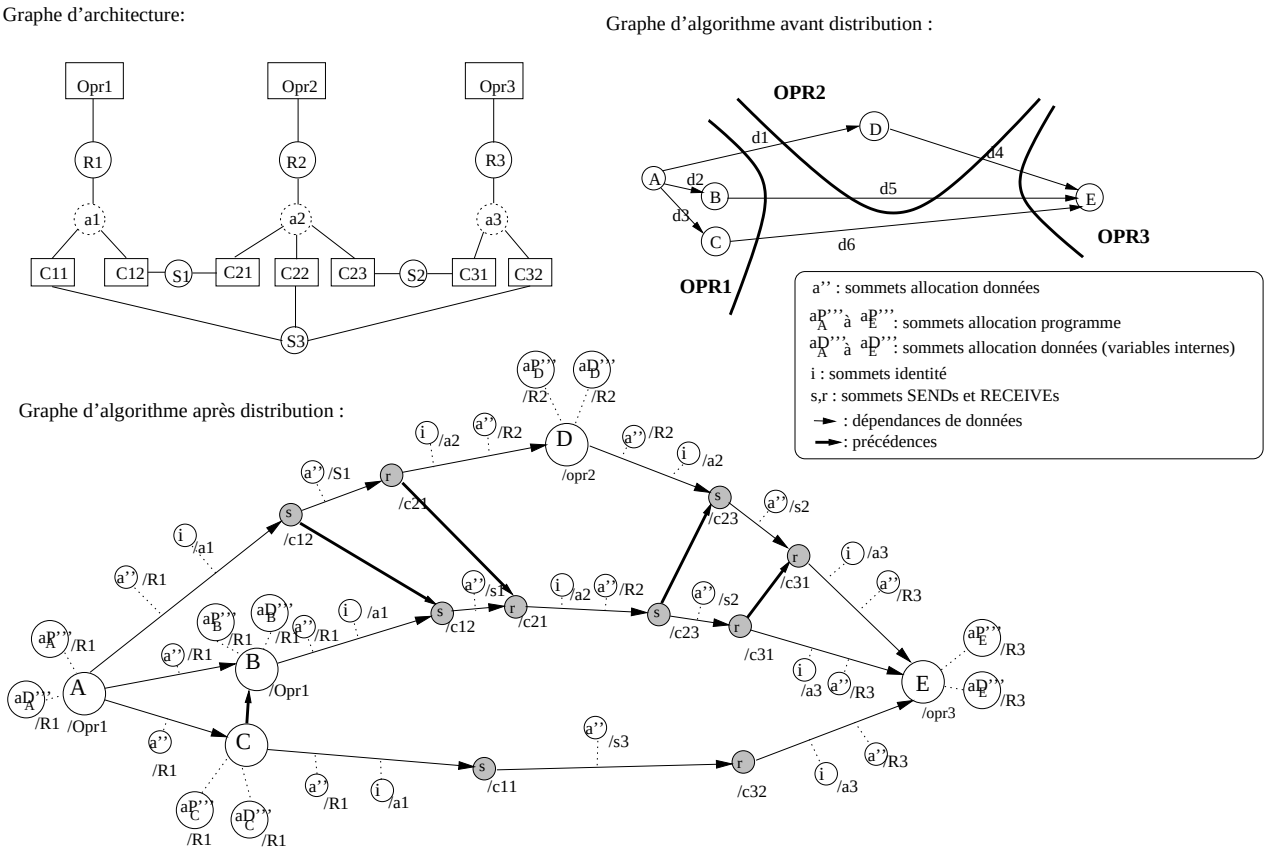


FIG. 3.19: *Exemple de graphe d'implantation après ordonnancement*

Chapitre 4

Optimisation

Sommaire

Les nouveaux modèles que nous avons introduit dans les précédents chapitres permettent d'effectuer un grand nombre de nouvelles optimisations. Ainsi nous allons étudier comment les nouveaux sommets SAM, RAM (avec ou sans arbitre) et bus (avec ou sans arbitre) permettent de minimiser l'utilisation de l'espace mémoire d'une application, d'améliorer les prédictions temporelles et par conséquent la qualité de l'optimisation de la durée d'exécution de l'application, mais aussi de minimiser la durée des communications qui peuvent être maintenant routées en parallèle.

Dans le chapitre précédent nous avons montré qu'à partir d'un graphe d'algorithme et d'un graphe d'architecture, il est possible de construire un très grand nombre de graphes d'implantation. Réaliser chaque implantation possible pour en mesurer les performances est inabordable, le nombre de solutions possibles peut en effet être très grand dans des cas réalistes (quelques dizaines d'opérations et quelques opérateurs). De plus, il se peut que l'architecture cible ne soit pas encore disponible au moment où l'on veut faire les comparaisons entre les implantations. La comparaison systématique étant impossible, il est nécessaire de construire un modèle prédictif des performances "pire cas" des implantations. Parmi l'ensemble des graphes d'implantation, nous recherchons ceux qui respectent les contraintes temps réel et minimisent la taille de l'architecture (utilisation optimisée des opérateurs, de la mémoire etc). La recherche de ces graphes d'implantation correspond à un problème d'allocation de ressource reconnu comme NP-difficile[70]. Dans le cadre du prototypage rapide, nous souhaitons obtenir une solution satisfaisante en un temps relativement court, même si ce n'est pas la solution optimale. Pour rechercher ces solutions, nous avons choisi d'utiliser

des heuristiques dites “gloutonnes” (sans retour arrière) basées sur du list-scheduling en utilisant les dates d’exécution des opérations.

4.1 Caractérisation

Le calcul des dates des opérations d’une implantation repose sur la durée d’exécution de chacune de ces opérations. Ces durées sont obtenues à partir de la caractérisation de chaque élément des graphes d’algorithme et d’architecture. Ces caractérisations sont obtenues soit par des mesures réelles [64, 31] (nous verrons dans la seconde partie, paragraphe 7.2.4, comment générer automatiquement un exécutif avec chronométrage pour faire ces mesures) soit, par exemple si l’architecture n’est pas disponible, par des estimations :

- chaque opérateur opr_i est caractérisé par l’ensemble des opérations $\lambda^{-1}(opr_i)$ (Cf. § 3.2.3.1), qu’il est capable d’exécuter. Il peut accéder à chaque RAM ou SAM connectée selon une bande passante maximale $BP_{max}(opr_i)$ dont le calcul sera exposé en 4.1.1,
- chaque opération de calcul et d’entrée-sortie o_j est caractérisée par sa durée d’exécution maximale (dans le pire des cas) sur chaque opérateur opr_i capable de l’exécuter, nous la notons : $\Delta(o_j, opr_i)$. Une opération est aussi caractérisée par la taille de sa mémoire programme notée $q'(\alpha_P(o_i))$, la taille de sa mémoire données $q'(\alpha_D(o_i))$ et sa bande passante moyenne d’accès à chaque mémoire connectée, notée $BP(o_i, ram_j)$ dont le calcul sera défini dans la section 4.1.1,
- chaque communicateur c_i est caractérisé par une bande passante maximale d’accès aux SAM et RAM qui lui sont connectées, notée $BP_{max}(c_i)$, et par un temps d’initialisation du transfert $\tau(c_i)$ (appelé aussi *start-up* [22]),
- chaque opération de communication est caractérisée par sa durée d’exécution qui est fonction du type et de la quantité de données à transférer ainsi que d’autres paramètres que nous verrons dans la section 4.1.2,
- chaque SAM, s_i, est caractérisée par sa taille $Q(s_i)$ et la valeur de la bande passante maximale selon laquelle il est possible d’y lire ou d’y écrire des données $BP_{max}(s_i)$,
- chaque mémoire RAM non partagée, m_i est caractérisée par sa taille $Q(m_i)$ et la valeur de la bande passante maximale selon laquelle il est possible d’y lire ou d’y écrire des données $BP(m_i)$,
- chaque RAM partagée m_i est aussi caractérisée par sa taille $Q(m_i)$, par contre chaque opérateur ou communicateur peut y lire ou écrire des données selon une bande passante qui dépend de la loi d’arbitrage de l’arbitre du Bus/Mux/Demux/Arbitre qui gère les accès à ces ressources partagées. Chaque Bus/Mux/Demux/Arbitre est donc caractérisé par sa loi d’arbitrage et les bandes passantes maximales des connexions à chaque opérateur ou communicateur comme nous le verrons dans la section 4.1.3,

4.1.1 Opérations de calcul et d’entrée-sortie

Nous avons vu dans le chapitre consacré au modèle d’algorithme qu’une opération correspond à l’encapsulation d’un ensemble d’instructions qu’exécutera un opérateur. Du point de vue de la caractérisation qui nous intéresse ici, la taille de cet ensemble (le grain) doit également (Cf. chapitre 2) être choisi de façon à encapsuler les détails architecturaux de l’opérateur (influences des mémoires caches et du pipeline des instructions et/ou de leurs opérandes). En effet, la prise en compte de ces détails est complexe et coûteuse

en temps d'optimisation, de plus on peut espérer que la variation relative de la durée d'exécution d'une séquence d'instruction soit inférieure à celle d'une instruction isolée, car les interactions entre instructions d'une même séquence ont tendance à se compenser (exploitation du pipeline, des registres internes rapides etc). Rappelons (cf.second chapitre) que les opérations sont codées de façon à accèder uniquement à de la mémoire RAM et non pas à de la SAM, qui en les couplant fortement rendrait inutilisable leur caractérisation individuelle.

Bandes passantes

Dans la prochaine section, afin de calculer la durée des opérations de communication, nous aurons besoin de connaître la bande passante utilisée par chaque opération pour accéder à chaque mémoire. Cette bande passante dépend du nombre d'accès mémoire réalisé par l'opération ainsi que de sa durée d'exécution. Dans le cas des opérations de calcul et d'entrée sortie, la durée d'exécution correspond à $\Delta(o_i, opr_j)$ (où $opr_j = \Pi(o_i)$), et le nombre d'accès mémoire est lié à :

- la quantité de mémoire programme de o_i : $q'(\alpha_P(o_i))$,
- la quantité de mémoire données locales de o_i : $q'(\alpha_D(o_i))$,
- la quantité de mémoire données communiquées, c'est à dire la somme des quantités de donnéess associée à chaque dépendance de données produite et consommée par o_i : $\sum_{\forall a_j / \gamma'(a_j)=o_i \quad ou \quad \gamma'^{-1}(a_j)=o_i} q'(a_j)$.

A partir de ces informations, il est possible de définir la bande passante moyenne requise par chaque opération o_i dans chaque mémoire m_k connectée à l'opérateur qui exécute o_i :

$$BP(o_i, m_k) = \frac{\sum_{\substack{\forall a_j / \Pi_{mem}(a_j) = m_k \quad et \\ (\gamma'(a_j) = o_i \quad ou \quad \gamma'^{-1}(a_j) = o_i)}} q'(a_j)}{\Delta(o_i, \Pi(o_i))} + \begin{cases} \frac{q'(\alpha_P(o_i))}{\Delta(o_i, \Pi(o_i))} & si \quad \Pi_P(\alpha_P(o_i)) = m_k \\ \frac{q'(\alpha_D(o_i))}{\Delta(o_i, \Pi(o_j))} & si \quad \Pi_D(\alpha_D(o_i)) = m_k \end{cases}$$

4.1.2 Communications

Dans le chapitre "Implantation", nous avons vu que les dépendances de données sont implantées en ajoutant différents sommets (allocations, identités et opérations de communication) distribués sur chaque route élémentaire composant les routes composées joignant les opérateurs des opérations productrices et consommatrices des dépendances de données. Nous allons maintenant étudier les caractéristiques temporelles des communications selon les différents types de routes existants, nous verrons en 4.2.3.5 l'algorithme qui permet de construire ces routes.

4.1.2.1 Route iso-opérateur

Dans le cas d'une route iso-opérateur, il n'y a pas d'opération de communication puisque les opérations productrices et consommatrices, lisent et écrivent leurs données dans la même RAM de la route. La durée de la communication est donc nulle.

Exemple 4.1.1 *La figure 4.1 a) présente le diagramme temporel d'implantation mono-opérateur d'un graphe d'algorithme composé de trois opérations A, B et C en dépendance de données. L'axe vertical de cette figure correspond au déroulement du temps. Les sommets du graphe d'implantation sont placés dans l'alignement vertical des sommets du graphe d'architecture qui leur sont associés. Ainsi, les cinq sommets allocation*

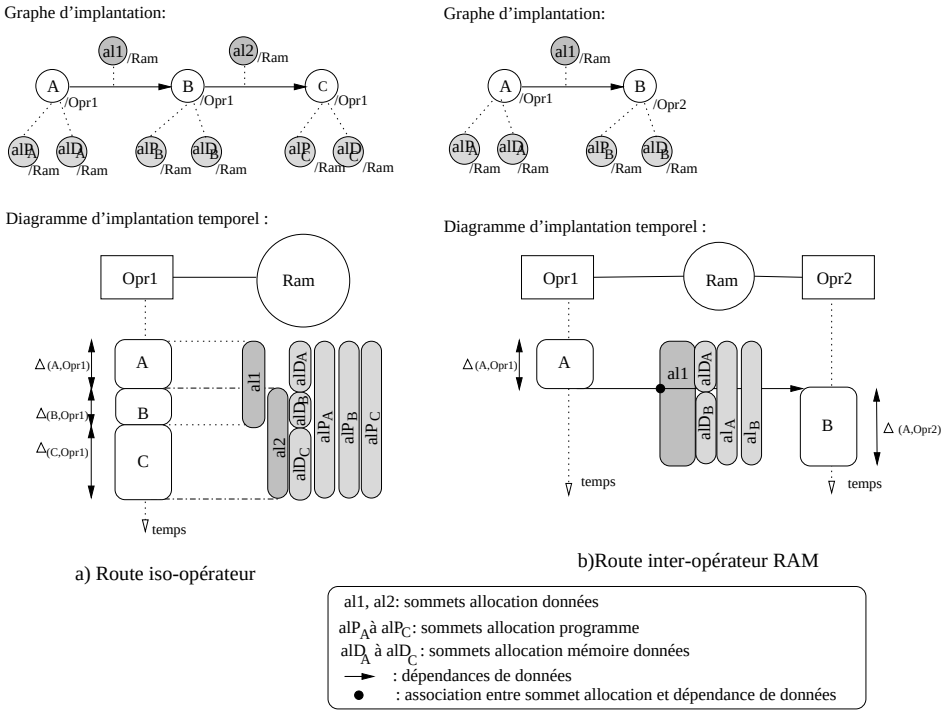


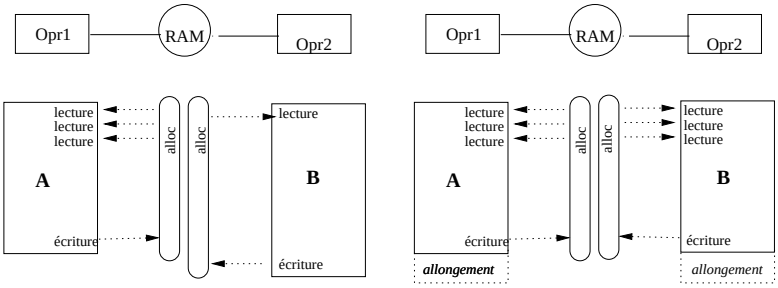
FIG. 4.1: *Communications par RAM*

($al1, al2$ et al_A à al_C) sont associés au sommet RAM et sont donc représentés en dessous. L'opérateur $Opr1$ exécute d'abord A pendant une durée $\Delta(A, Opr1)$ puis il exécute B pendant une durée $\Delta(B, Opr2)$. Le sommet allocation $al1$, qui implante la dépendance de données entre A et B à $al1$, a une durée de vie égale à la somme de ces deux opérations.

4.1.2.2 Route inter-opérateurs : mémoire RAM partagée

Ce type de route ne possède pas de communicateur, les opérations productrices et consommatrices accèdent successivement à la même RAM partagée (Cf. exemple b) de la 4.1) . Comme dans le cas des routes iso-opérateur la durée de la communication est nulle. Cependant, parce que la RAM est partagée avec d'autres opérateurs, il se peut que ces derniers exécutent des opérations qui accèdent à la RAM simultanément. Dans ce cas, selon l'arbitrage de la RAM, la bande passante d'accès à la RAM par les opérateurs peut être restreinte, ce qui peut avoir pour conséquence d'allonger la durée d'exécution des opérations exécutées par ces opérateurs.

Calculer précisement l'allongement de l'exécution d'une opération en fonction de la bande passante des mémoires dans lesquelles elle accède est un problème complexe. Il se peut par exemple que deux opérations exécutées simultanément par deux opérateurs connectés à une même RAM partagée ne se ralentissent pas mutuellement si elles n'accèdent pas au même instant à la mémoire (Cf. exemple a) de la figure 4.2). Par contre, si des opérations identiques sont exécutées par des opérateurs identiques connectés à une même RAM, elles accèderont à cette RAM au même instant et se ralentiront donc mutuellement (Cf. exemple b) de la figure 4.2). Nous avons choisi dans un premier temps, de négliger cet allongement et d'utiliser uniquement le calcul des bandes passantes pour obtenir la durée des opérations de communication.



a) exécution parallèle sans accès simultané à la RAM

b) exécution parallèle avec accès simultanés à la RAM

FIG. 4.2: *Interactions entre opérations*

4.1.2.3 Route inter-opérateurs : RAM et communicateurs

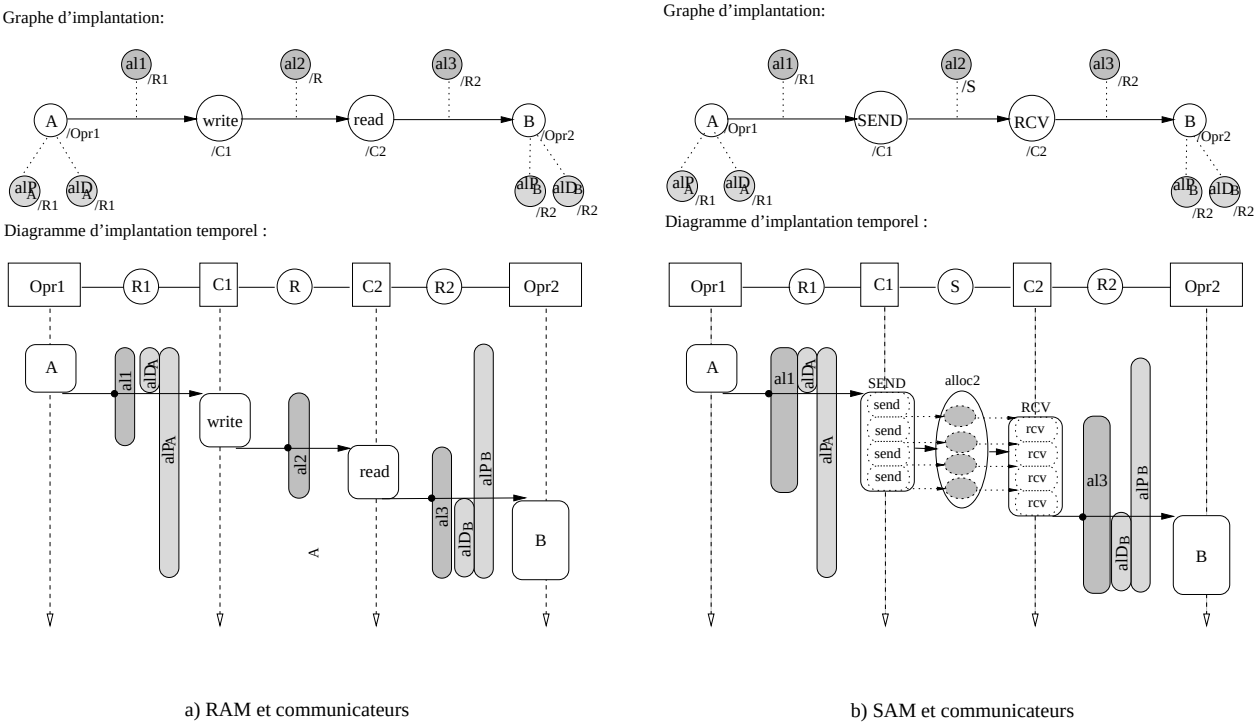
La durée de communication d'une dépendance de données d_i, sur une route élémentaire composée de communicateurs se partageant une RAM, est égale à la somme des durées des opérations de communication d'écriture (`write`) et de lecture (`read`) dans la RAM partagée. La durée de chacune de ces opérations est fonction du temps d'initialisation du communicateur (τ) qui les exécute, de la quantité de données ($q(d_i)$) à transférer, du type des données ($type(d_i)$), et de la bande passante de la route. Cette dernière correspond au minimum des bandes passantes de chaque élément (RAM, SAM, communicateur, Bus/Mux/Demux/Arbitre) qui composent la route. C'est en effet le sommet le plus lent qui impose sa bande passante.

Exemple 4.1.2 *La figure 4.3 a) présente le graphe d'implantation d'un graphe d'algorithme composé de deux opérations A et B en dépendance de données. La durée du sommet allocation $al1$ associé à $R1$ est égale à la somme des durées de l'opération de calcul A et de l'opération de communication $write$ qui lit les données produites par A pour les copier sur la RAM partagée par les communicateurs. Rappelons (Cf. § 1.2.2.9, 29) que ce type de communication par RAM partagée et communicateurs est préférée à celle des exemples précédents lorsque la bande passante de la RAM partagée est beaucoup plus faible que celle des opérateurs.*

4.1.2.4 Route inter-opérateurs : SAM et communicateurs

Nous avons vu (Cf. 81) que la taille des mémoires SAM est souvent faible et implique le découpage des données en paquets (de taille égale à celle des SAM) écrits et lus séquentiellement (car ce type de mémoire impose un ordre entre lecture et écriture). Dans le chapitre précédent (Cf. figure 3.9, page 82) nous avons factorisé les différentes écritures et lectures qui réalisent une communication, à travers les sommets `SEND` et `RECEIVE` . Du point de vue de l'exécution de ces sommets, on peut noter qu'il y a recouvrement temporel entre le `SEND` et le `RECEIVE` comme l'indique le schéma b) de la figure 4.3. Si l'on néglige les durées de transfert des premiers et derniers paquets (quand le nombre des paquets est grand), on peut considérer que les sommets `SEND` et `RECEIVE` se recouvrent complètement, leur exécution peut être considérée comme simultanée sur leur communicateur respectif, et donc leur durée d'exécution considérée égale, et par conséquence leur date de début et de fin aussi.

Comme pour les opérations `read` et `write` , la durée des opérations `SEND` et `RECEIVE` est fonction du temps d'initialisation du communicateur, de la quantité de données à transférer, de leur type, et de la bande passante de la route.



a) RAM et communicateurs

b) SAM et communicateurs

alD $_A$ et alD $_B$ sommets allocation mémoire données
alP$_A$ et alP$_B$ sommets allocation mémoire programme
al1 à al3 : sommets allocation dépendance données
→ : dépendances de données
• : association entre dépendance de données et allocation

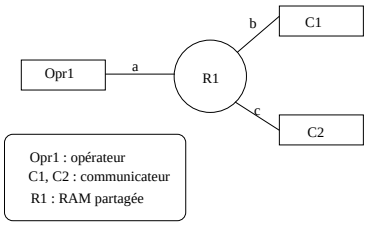
FIG. 4.3: *Communications par communicateurs*

Exemple 4.1.3 *La figure 4.3 b) présente le graphe d'implantation d'un algorithme composé de deux opérations A et B sur une architecture composée de deux opérateurs connectés chacun à des communicateurs ($C1, C2$) eux même connectés par une SAM (S). Sur cet exemple nous avons défactorisé les opérations $SEND$ et $RECEIVE$ de façon à mettre en évidence le recouvrement temporel entre ces deux opérations de communication. Si il n'y a pas de conflit d'arbitrage, la durée de communication d'une dépendance de données d_i est donc égale à :*

$$d = \frac{Q(d_i)}{\min(BP(ram1), BP(com1), BP(sam), BP(com2), BP(ram2)} + \max(\tau(com1), \tau(com2))$$

4.1.3 Arbitrage

Les arbitres des Bus/Mux/Demux/Arbitre et des RAM partagées sont utilisés pour gérer les conflits d'accès aux mémoires par les opérateurs et les communicateurs. Ces derniers font des requêtes de bande passante (soit pour lire, soit pour écrire des données dans les mémoires qui leur sont connectées) et l'arbitre alloue de la bande passante à chaque opérateur et communicateur. A chaque instant physique, la bande passante qu'obtient chaque opérateur et/ou communicateur est dépendante des autres bande passantes ainsi qu'à la priorité de chaque opérateur et communicateur. Par exemple, pour étudier l'arbitrage d'une RAM (Cf. figure 4.4) partagée par un opérateur ($Opr1$) et deux communicateurs ($C1, C2$), nous avons encodé ses caractéristiques sous la forme d'une table qui indique, la bande passante totale de l'arbitre (2Mb/s), les bandes passantes maximales de chaque connexion (1 Mb/s par connexion) et les relations de priorité entre chaque connexion : les arcs B et C ont même priorité qui est inférieure à celle de l'arc A.



R1	BPmax totale 2Mb/s		
arc	a	b	c
BPmax	1Mb/s	1Mb/s	1Mb/s
priorité	1	0	0

FIG. 4.4: *Arbitrage d'une RAM partagée*

Avec cet arbitrage, l'opérateur $opr1$, connecté par l'arc a prioritaire, obtiendra toujours une bande passante maximale de 1Mb/s. Si l'un des deux communicateurs doit accéder à la mémoire, il obtiendra une bande passante de 1Mb/s (que l'opérateur accède ou non à la RAM en parallèle). Par contre si les deux communicateurs accèdent simultanément à la RAM ils obtiendront une bande passante de 1Mb/s chacun, si l'opérateur n'y accède pas, et 0.5Mb/s si l'opérateur y accède en parallèle avec la bande passante maximale. Par la suite, ces bandes passantes vont permettre de prédire les durées de communication.

4.1.4 Calcul de dates

Après avoir caractérisé chaque opération de calcul et de communication par une durée d'exécution sur un opérateur ou un communicateur, il est possible, grâce à la relation d'ordre associée au graphe d'implantation, de déterminer les dates de début et de fin de chaque opération. Ces dates sont à la base de nos heuristiques d'optimisation. Nous allons définir des dates par rapport au début de l'exécution, mais aussi des dates depuis la fin de l'exécution. A partir de ces deux types de date nous pourrons ensuite définir la *flexibilité d'ordonnancement* d'une opération qui est utilisée lors de l'optimisation.

4.1.4.1 Dates associées à un graphe d'algorithme

Pour calculer la flexibilité et la pénalité d'une opération, il nous faudra calculer les dates associées à chaque opération d'un graphe d'algorithme non implanté. Étant donné que ces opérations ne sont pas associées à des opérateurs, leur durée d'exécution n'est pas définie ce qui rend impossible le calcul de date. Pour palier à cela, nous choisissons d'utiliser une approximation de leur durée d'exécution basée sur leur durée d'exécution moyenne. Cette durée d'exécution approximée, notée $\delta_{app}(o_i)$, est définie par :

$$\Delta_{app}(o_i) = \frac{1}{n_i} \sum_{j=1}^{n_i} \Delta(o_i, p_j)$$

et n_i désigne le nombre de processeurs pouvant exécuter l'opération o_i, c'est à dire $n_i = card(\lambda(o_i))$.

Remarque 30 *Dans le cas d'une architecture homogène, la durée d'exécution de chaque opération est identique sur tous les opérateurs.* Δ_{app} *n'est alors plus une approximation mais la valeur exacte.*

Dates au plus tôt définies depuis le début La date de fin au plus tôt $E(o_i)$ (pour End) d'une opération o_i correspond à sa date de début au plus tôt, notée $S(o_i)$ (pour $Start$) sommée avec sa durée d'exécution moyenne $\Delta_{app}(o_i)$:

$$E(o_i) = S(o_i) + \Delta_{app}(o_i)$$

Les dépendances de données entre les opérations d'un graphe flots de données induisent des précédences d'exécution. L'exécution d'une opération o_i ne peut donc avoir lieu avant la fin de l'exécution de tous ses prédécesseurs $\Gamma^{-1}(o_i)$. La date de début au plus tôt d'une opération o_i correspond donc à la plus grande date de fin au plus tôt de ses prédécesseurs (Cf. exemple figure 4.5) :

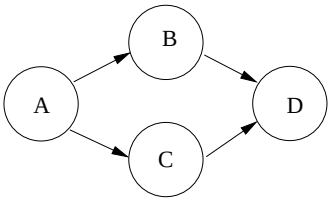
$$S(o_i) = \begin{cases} 0 \quad si \quad \Gamma^{-1}(o_i) = \emptyset \\ \max\limits_{\forall o_j \in \Gamma^{-1}(o_i)} E(o_j) \quad sinon \end{cases}$$

Remarque 31 *La date de début au plus tôt est parfois appelée* $ASAP$ *"As Soon As Possible",* $AEST$ *"Absolute Earliest Start Time" [60] ou encore* ST *"Starting Time" [103].*

Chemin critique A partir de ces dates il est possible de calculer la longueur du chemin critique $\mathcal{R}$ du graphe d'algorithme, c'est à dire la durée du chemin le plus long de ce graphe (Cf. exemple figure 4.5) :

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

Graphe d'algorithme :



Durées moyennes d'exécution des opérations :

$\Delta_{app}(A)$	$\Delta_{app}(B)$	$\Delta_{app}(C)$	$\Delta_{app}(D)$
1	4	1	1

Diagramme temporel du graphe d'algorithme :

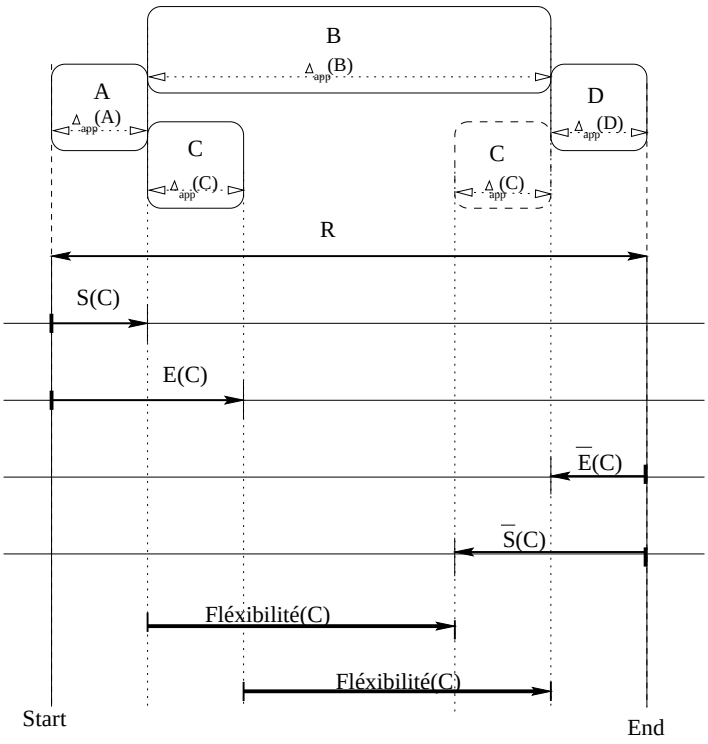


FIG. 4.5: *Dates des opérations d'un algorithme*

Exemple 4.1.4 *La figure 4.5 présente un graphe d'algorithme et son diagramme temporel qui est construit à partir des relations de précédences entre les opérations du graphe d'algorithme et des durées d'exécution moyenne de chaque opération du graphe d'algorithme. Les opérations sont représentées par des rectangles dont la largeur est proportionnelle à leur durée moyenne d'exécution. Les opérations B et C ne sont pas en dépendance de données, elle peuvent être exécutées en parallèle et sont donc représentées sur des axes horizontaux parallèles. Comme ces deux opérations sont en dépendance de données avec A, leur exécution ne peut commencer qu'après la fin de A, c'est pourquoi les rectangles représentants B et C sont placés à droite du rectangle qui représente A. De même, comme l'exécution de D ne peut avoir lieu qu'après la fin de B et C, le rectangle qui représente E est placé à droite des rectangles B et C.*

Dates au plus tard définies depuis la fin Dans les précédents paragraphes, nous avons défini des dates par rapport à une origine qui est le début de l'algorithme, c'est à dire par rapport aux opérations sans prédécesseur. Ici nous définissons des dates au plus tard par rapport à la fin de l'algorithme (i.e. les opérations sans successeur), cela permettra par la suite de ne manipuler que des dates positives. La date de fin au plus tard depuis la fin $\bar{S}$ (la barre signifie à la fois "au plus tard" et "depuis la fin") d'une opération o_i, correspond à la plus grande date de début au plus tard depuis la fin $\bar{E}$ de ses successeurs (Cf. exemple figure 4.5) :

$$\bar{E}(o_i) = \begin{cases} O & si \quad \Gamma(o_j) = \emptyset \\ \max\limits_{o_j \in \Gamma(o_j)} \bar{S}(o_j) & sinon \end{cases}$$

La date de début au plus tard depuis la fin d'une opération correspond à sa date de fin depuis la fin

sommée avec sa durée d'exécution moyenne :

$$\bar{S}(o_i) = \bar{E}(o_i) + \Delta_{app}(o_i)$$

4.1.4.2 Dates associées à un graphe d'implantation

Dates au plus tôt définies depuis le début La date de fin au plus tôt depuis le début $E(o_i)$ (pour End) d'une opération o_i exécutée par un opérateur $p_j = \Pi(o_i)$ correspond à sa date de début, notée $S(o_i)$ (pour $Start$) sommée avec sa durée d'exécution $\Delta(o_i, \Pi(o_i))$:

$$E(o_i) = S(o_i) + \Delta(o_i, \Pi(o_i))$$

Pour le calcul de date au plus tôt, par rapport au graphe d'algorithme non implanté, il faut en plus tenir compte de la date de fin de la dernière opération exécutée sur l'opérateur de o_i, c'est à dire $\Gamma'^{-1}(o_i)$:

$$S(o_i) = \begin{cases} 0 \quad si \quad \Gamma^{-1}(o_i) = \emptyset \quad et \quad \Gamma'^{-1}(o_i) = \emptyset \\ \max(\max\limits_{\forall o_j \in \Gamma^{-1}(o_i)} E(o_j), E(\Gamma'^{-1}(o_i))) \quad sinon \end{cases}$$

Dates au plus tard définies depuis la fin Symétriquement, le calcul de date par rapport à la fin devient :

$$\bar{E}(o_i) = \begin{cases} O \quad si \quad \Gamma(o_j) = \emptyset \quad et \quad \Gamma'(o_j) = \emptyset \\ \max(\max\limits_{o_j \in \Gamma(o_j)} \bar{S}(o_j), \bar{S}(\Gamma'(o_j))) \quad sinon \end{cases}$$

La date de début depuis la fin d'une opération correspond à sa date de fin depuis la fin sommée avec sa durée d'exécution :

$$\bar{S}(o_i) = \bar{E}(o_i) + \Delta(o_i, \Pi(o_i))$$

4.1.4.3 Dates des sommets allocation et identité

La date de début d'utilisation des sommets allocation (O''_{alloc}) et identité (O''_{ident}) qui implantent une dépendance de données, correspond à la date de début de l'opération productrice de la dépendance de données que ces sommets implantent. La date de fin correspond à la plus grande date de fin parmi les opérations consommatrices de la dépendance qu'ils implantent :

$$\forall a''_i \in O''_{alloc}, \begin{cases} S(a''_i) = S(\gamma^{-1}(\alpha^{-1}(a''_i))) \\ E(a''_i) = \max\limits_{\forall o_j \in \gamma(\alpha^{-1}(a''_i))} E(o_j) \end{cases}$$

La durée d'un sommet allocation mémoire programme (O'''_{allocP}) est égale à celle de la durée totale de l'application, ses dates sont donc toujours :

$$\forall a'''_i \in O'''_{allocP}, \begin{cases} S(a'''_i) = 0 \\ E(a''_i) = \mathcal{R} \end{cases}$$

La durée d'un sommet allocation mémoire données (O'''_{allocD}) est égale à celle de l'opération correspondante :

$$\forall a'''_i \in O'''_{allocD}, \begin{cases} S(a'''_i) = S({\alpha_D}^{-1}(a'''_i)) \\ E(a''_i) = E({\alpha_D}^{-1}(a'''_i)) \end{cases}$$

4.1.5 Flexibilité d'ordonnancement

La date de début d'une opération, comme son nom l'indique, est la date de début minimale d'une opération (i.e. il est impossible que l'exécution de cette opération ait lieu avant cette date en respectant l'ordre partiel indiqué par les dépendances). La date de début au plus tard depuis la fin d'une opération est une date après laquelle l'exécution de l'opération engendrerait l'allongement du chemin critique $\mathcal{R}$ du graphe. La longueur de l'intervalle entre ces deux dates est appelée flexibilité d'ordonnancement [74, 66] (nommée parfois mobilité ou marge d'ordonnancement). Si la date de début effective d'exécution de l'opération a lieu entre ces deux dates, cela ne modifie pas la longueur du chemin critique (Cf. exemple figure 4.5) :

$$F(o_i) = \mathcal{R} - \bar{S}(o_i) - S(o_i) = \mathcal{R} - \bar{E}(o_i) - E(o_i)$$

Les opérations qui appartiennent au chemin critique n'ont pas de flexibilité (par exemple A, B, D sur le graphe de la figure 4.5), ces opérations sont dites *critiques*. Leur date de début au plus tôt depuis le début est équivalente à celle de leur date de début au plus tard depuis la fin :

$$S(o_i) = \mathcal{R} - \bar{S}(o_i) \quad et \quad E(o_i) = \mathcal{R} - \bar{E}(o_i)$$

4.1.6 Mémoire RAM

4.1.6.1 Capacité

Chaque fois qu'un sommet allocation $a \in O''_{alloc} \cup O'''_{allocP} \cup O'''_{allocD}$ est associé à un sommet RAM ram_j , la capacité $Q(ram_j)$ de cette mémoire est diminuée de la quantité $q'(a)$ pendant la durée du sommet allocation.

4.1.6.2 Diagramme mémoire

Dans cette section nous dressons un diagramme à la fois spatial et temporel d'utilisation de la mémoire, il permettra d'en optimiser son utilisation dans la section 4.3. Pour la construction de ce diagramme, nous devons poser certaines hypothèses quant à la génération de code sous-jacente à l'allocation mémoire. Ainsi, pour simplifier les explications, nous allons considérer, car c'est souvent le cas, que chaque allocation mémoire programme et mémoire données (O'''_{allocP} et O'''_{allocD}) nécessaires à l'exécution d'une opération, sont toujours effectuées dans des zones différentes (appelées *zone programme* et *zone donn´ees locale*) de celle des sommets allocation qui implantent les dépendances de données (appelée *zone donn´ees communiqu´ees*). Ainsi, dans les diagrammes suivants, nous considérons que la zone mémoire données communiquées est allouée selon les adresses croissantes depuis le début de la mémoire par les sommets allocation des dépendances de données (O''_{alloc}), et allouée selon des adresses décroissantes depuis la dernière adresse de cet espace pour les sommets allocation mémoire programme (O'''_{alloc}) de la zone programme et pour les sommets allocations (O'''_{allocD}) de la zone données locales .

Exemple 4.1.5 *La figure 4.6 présente un graphe d'algorithme très simplifié, un exemple de graphe d'implantation de ce graphe d'algorithme, le diagramme temporel de cette implantation, et le diagramme temporel d'utilisation mémoire correspondant à la RAM $R1$. Le sommet A est producteur de deux dépendances de données. La première est consommée par B, la seconde est consommée par B et C (elle diffuse). La date d'utilisation au plus tôt de l'espace mémoire réservé par le sommet allocation qui implante la dépendance AB (al_{AB}), correspond à la date de début d'exécution de l'opération A, soit $S(A)$. La date de fin d'utilisation au plus tard de cet espace correspond à la date de fin de B, soit $E(B)$. Comme l'allocation mémoire*

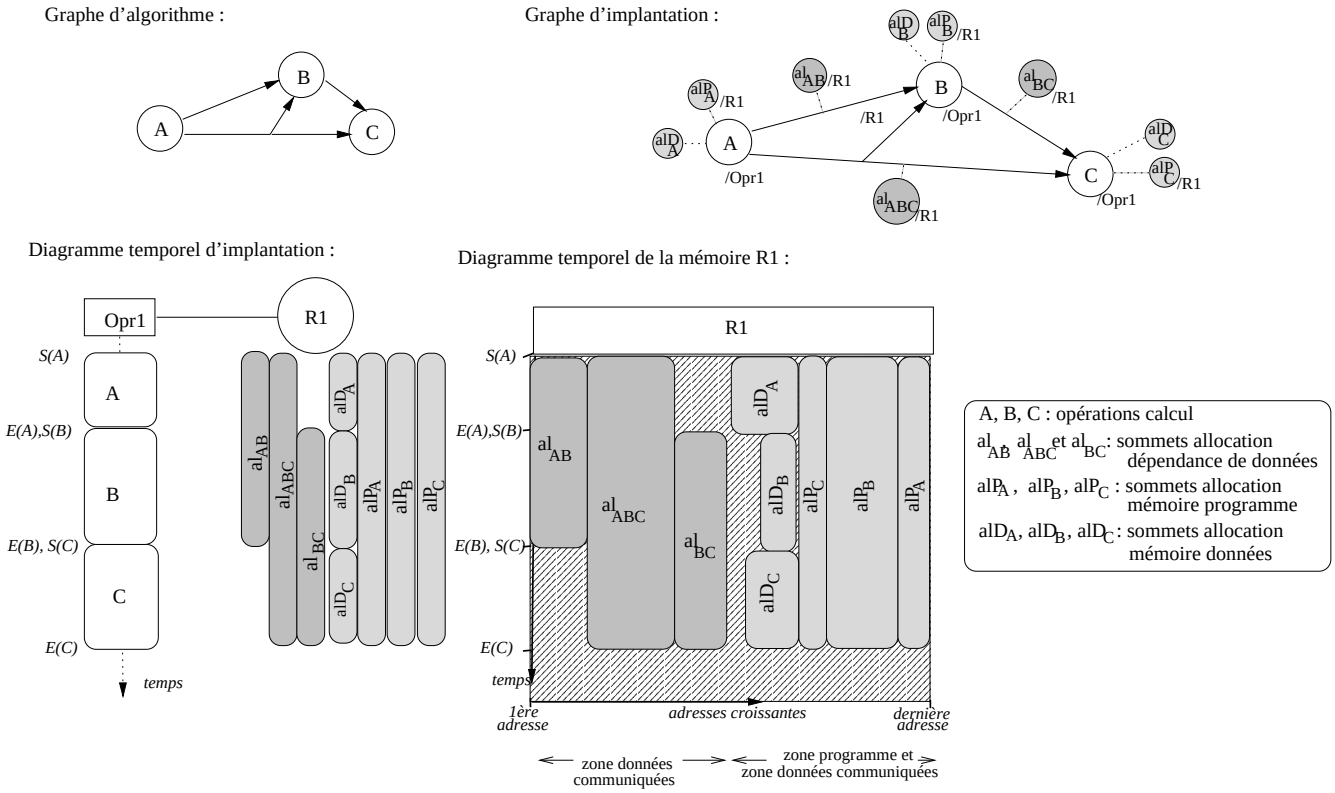


FIG. 4.6: *Diagramme d'utilisation de la mémoire*

programme doit être permanente, la durée de chacun de ces sommets (alP_A, alP_B et alP_C) est égale à la durée de l'application. La partie hachurée de la mémoire correspond à la partie non encore allouée (libre) par l'implantation.

Comme les sommets allocation n'interviennent pas dans le calcul des dates des opérations, et qu'ils induisent une certaine complexité graphique, il est possible de ne pas les représenter sur les diagrammes temporels d'implantation. Cependant, pour mettre en correspondance les dates des opérations et celles d'utilisations des sommets allocation, il est possible d'encapsuler les diagrammes temporels d'utilisation de chaque RAM dans les diagrammes temporels d'implantation comme nous l'avons fait dans la figure 4.7 de l'exemple suivant.

Exemple 4.1.6 *La figure 4.7 présente un exemple de graphe d'implantation temporel incluant les diagrammes temporels d'utilisation des mémoires. Nous constatons sur cet exemple que la RAM $R1$ est presque entièrement utilisée. Nous verrons comment il est possible de libérer de la mémoire dans le paragraphe 4.3 consacré à l'optimisation de l'usage de la mémoire.*

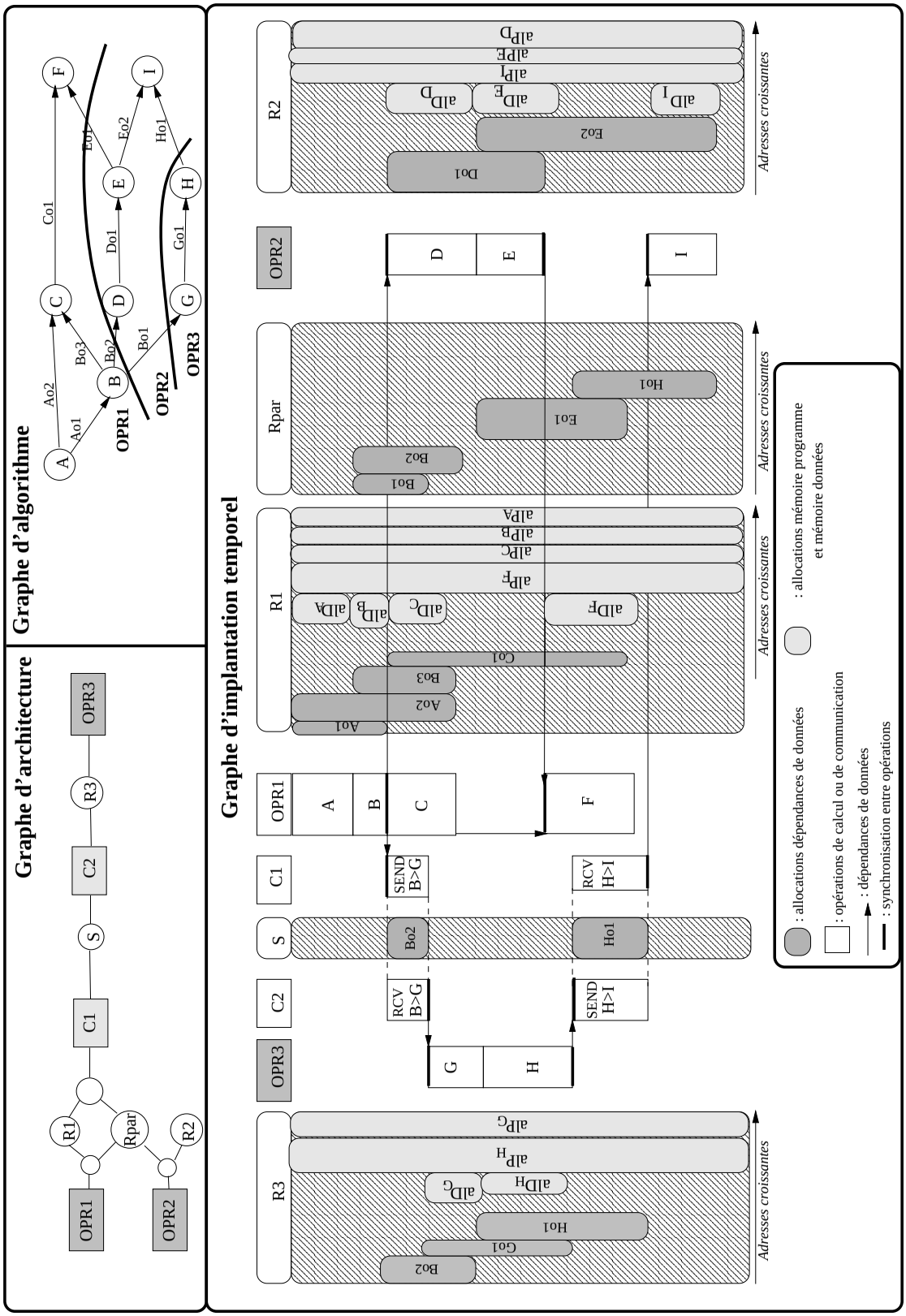


FIG. 4.7: *Exemple détaillé*

4.2 Optimisation de la latence égale à la cadence

4.2.1 Méthodes de résolution

Dans le chapitre précédent nous avons donné les règles qui permettent de construire l'ensemble des graphes d'implantation valides d'un graphe d'algorithme sur un graphe d'architecture. Dans la précédente section nous avons donné les règles qui permettent de calculer les dates d'exécution des opérations de ces graphes. Dans cette section nous allons donner les méthodes qui permettent de déterminer, parmi ces implantations, celles dont la latence respecte la contrainte temps réel de l'application. Rappelons que la latence d'une application désigne le temps qui s'écoule entre la première opération d'entrée-sortie du graphe (capteur) et la dernière (actionneur) et que la cadence est le temps qui s'écoule entre deux opérations d'entrée-sortie identique. La latence est donc la durée totale d'une seule itération du graphe d'implantation, elle correspond à la longueur R du chemin critique de ce graphe. Déterminer ce graphe d'implantation est un problème d'allocation de ressources reconnu comme un problème NP-complet[90, 36]. Explorer exhaustivement toutes les solutions n'est donc pas abordable étant donné la complexité des applications temps réel que nous devons traiter. En effet, bien qu'il existe différentes méthodes exactes [39, 16, 86, 10] conduisant à la solution optimale, elles sont toutes limitées à des problèmes de petite taille car le temps d'exploration est très long en pratique. De plus, elles reposent souvent sur des hypothèses incompatibles avec le contexte des applications temps réel embarquées (comme par exemple des ressources illimitées).

Pour résoudre ce problème, nous devons donc utiliser des méthodes approchées appelées aussi *heuristiques*. Ces méthodes ont l'intérêt de fournir rapidement une solution sous-optimale acceptable. On distingue deux grandes classes de méthodes[83] : les approches dites *gloutonnes* et les méthodes de *voisinage*.

Méthodes gloutonnes

Les méthodes gloutonnes construisent la solution itérativement en prenant à chaque étape une décision (distribution et/ou ordonnancement) qui n'est jamais remise en cause par la suite (pas de retour arrière, "back-tracking"). Les *algorithmes de clustering* et les *algorithmes de liste* font partie des méthodes gloutonnes. Les premiers [105, 60, 55] construisent l'implantation en deux étapes : distribution puis ordonnancement alors que les algorithmes de liste effectuent la distribution et l'ordonnancement simultanément (on dit qu'ils sont *one-stage* [104] alors que les premiers sont qualifiés de *multi-stage*).

- Les algorithmes de clustering partitionnent l'algorithme par *regroupement des op´erations* de façon à minimiser les transferts de données entre les opérations et à réduire ainsi la longueur du chemin critique. Quand toutes les opérations ont été traitées, les groupes (cluster) qui ont été construits sont associés aux processeurs de l'architecture. Les clusters sont alors ordonnancés. Enfin, les dépendances entre clusters sont ensuite transformées en communications distribuées sur les liens de communication qui composent l'architecture.

- Les algorithmes de liste utilisent une règle de priorité associée à une fonction de coût pour classer les opérations dans une liste. Cette liste est ensuite utilisée pour distribuer et ordonnancer les opérations selon leur priorité décroissante. La liste peut être statique ou dynamique. Lorsqu'elle est statique la priorité associée à chaque opération n'est jamais recalculée alors que dans le cas de liste dynamique, les priorités sont recalculées à chaque étape de l'heuristique. Les listes dynamiques permettent donc de tenir compte des modifications de priorité que peut entraîner une implantation partielle sur les opérations non encore implantées.

Méthodes de voisinages

Les méthodes de voisinages cherchent à améliorer une solution existante éventuellement issue d'une méthode gloutonne. Elles sont classées en deux catégories selon qu'elles reposent sur une méthode de recherche locale ou une méthode de recherche globale.

- Les méthodes de recherche locale consistent à rechercher dans un espace local de meilleures solutions. Par exemple l'heuristique FAST [61] est basée sur l'étude des opérations dites "bloquantes" (quand on enlève une telle opération d'un processeur, les opérations qui lui succèdent peuvent commencer plus tôt) qui constituent le voisinage de recherche pour la fonction de coût.

- Les méthodes de recherche globale (le recuit simulé[22], les algorithmes génétiques[83, 27], le Tabou[83]) consistent à rechercher globalement dans l'ensemble des solutions possibles. Par exemple, le recuit simulé est une méthode stochastique qui tire au sort une solution voisine à chaque itération pour en évaluer son coût. Si cette solution est meilleure, elle est conservée. Sinon on calcule sa probabilité d'acceptation pour décider de la conserver ou de la rejeter. Cet heuristique permet de sortir du minimum local de la fonction de coût afin d'atteindre éventuellement son minimum global.

Conclusion

Les méthodes de liste et de clustering sont simples et rapides à mettre en œuvre, de plus elles donnent rapidement des résultats ce qui convient bien au prototypage rapide. Elles ne garantissent cependant pas d'obtenir le minimum global. Les méthodes de voisinage de type recuit simulé sont capables de fournir le minimum global, cependant elles sont plus longues à mettre en œuvre (elles nécessitent de régler de nombreux paramètres) et ont un temps d'exécution plus important. Des comparaisons expérimentales [74, 83] entre les méthodes de liste et de voisinage global ont montré que les heuristiques de liste fournissent dans 70% des cas d'aussi bons résultats que les méthodes de recuit simulé en un temps 100 fois plus petit. Dans la suite de ce chapitre nous allons présenter les heuristiques gloutonnes de type liste dynamique que nous avons choisi pour construire le graphe d'implantation temps réel d'un algorithme et d'une architecture dans lequel la latence, égale à la cadence, est minimisée.

4.2.2 Heuristique proposée

Comme pour toutes les heuristiques gloutonnes basées sur une méthode de liste ("*list scheduling*"), il nous faut définir des règles de construction de liste et une fonction de coût qui prend en compte la durée des opérations du graphe de l'algorithme et la durée des communications quand des opérations en dépendances de données sont distribuées sur des opérateurs différents.

La fonction de coûts, que nous noterons σ (Cf. 4.2.2.2) et les règles de construction de la liste sont basées sur [66], et formalisées dans [102] , elles ont été adaptées aux architectures hétérogènes dans[42]. Nous les enrichissons ici de façon à faire intervenir le coût des mémoires qui ne faisait pas parties du modèle d'architecture utilisé dans ces précédents travaux. La distribution et l'ordonnancement des communications sont traités de façon à optimiser les communications (routage parallèle, réutilisation de communication lors de diffusion).

4.2.2.1 Principe

A chaque étape, l'heuristique de distribution/ordonnancement sélectionne une opération dans une liste d'opérations dites *candidates*, puis distribue et ordonnance cette opération sur un opérateur préalablement

choisi. Après chaque distribution/ordonnancement, la liste de candidats est remise à jour.

Remarque 32 *Afin de simplifier la lecture, par la suite nous utiliserons uniquement le terme ordonnancement pour désigner la distribution et l'ordonnancement qui s'effectuent toujours en même temps dans nos heuristiques.*

Algorithme simplifié

- Routage : construction de toutes les routes les plus courtes entre chaque couple d'opérateurs,
- Au départ, la liste contient donc les opérations qui n'ont pas de prédécesseur. En effet, à chaque itération la liste de candidats ne doit contenir que les opérations dites *ordonnan¸cables*. Une opération est ordonnançable quand tout ses prédécesseurs sont distribués et ordonnancés. Ceci permet de garantir que la construction de l'ordonnancement sur chaque opérateur, respecte l'ordre partiel d'exécution entre les opérations,
- la sélection d'une opération s'effectue en trois étapes :
    1. pour chaque opération de la liste, rechercher le "meilleur opérateur" (i.e. celui qui minimise la fonction de coût, Cf. § 4.2.2.2) pour chaque opération candidate. Pour cela, on évalue le coût de l'ordonnancement de chaque opération candidate sur chaque opérateur, en prenant en compte les communications induites sur chaque route par le choix de distribution,
    2. restreindre la liste de candidats selon la valeur de leurs dates de début relatives de façon à éviter que les opérateurs soient trop souvent inoccupés,
    3. choisir dans la liste restreinte l'opération qui est la plus urgente à ordonnancer grâce à cette même fonction de coût (Cf. § 4.2.3.2),
- la mise à jour de la liste consiste à :
    1. en retirer l'opération qui vient d'y être ordonnancée,
    2. y ajouter tous les successeurs qui deviennent ordonnançables suite à l'ordonnancement de cette opération.

Algorithme détaillé L'algorithme suivant décrit tous les détails régissant le fonctionnement de cette heuristique. Nous avons pris pour convention que l'exposant entre parenthèses désigne l'étape de l'heuristique, ainsi $O_{cand}^{(n)}$ désigne l'ensemble des opérations candidates à l'étape n.

- Routage : construction de toutes les routes les plus courtes entre chaque couple d'opérateurs (Cf. § 4.2.3.5)

- Initialiser la liste de candidats avec les opérations sans prédécesseur, ou dont les seuls prédécesseurs sont des opérations de type retard ou constante (Cf. § 4.2.3.3 et § 4.2.3.4): $O_{cand}^{(1)} = \{o_i \in O / \Gamma^{-1}(o_i) = \emptyset \quad ou \quad \Gamma^{-1}(o_i) \in O'_{\$} \quad ou \quad \Gamma^{-1}(o_i) \in O'_{const}\}$

- Tant que la liste n'est pas vide (tant que $O_{cand}^{(n)} \neq \emptyset$) faire :

  - Pour chaque opération candidate o_i, rechercher le meilleur opérateur p_i^{best}. Pour cela on ordonnance et dé-ordonnance chaque opération candidate o_i sur chacun des opérateurs capable de les exécuter $\lambda(o_i)$, en construisant et détruisant chaque fois les communications des dépendances de données entre o_i et tous ses prédécesseurs $\Gamma^{-1}(o_i)$ (Cf. §4.2.3.5), on obtient ainsi des couples (o_i, p_i^{best}) tel que (Cf. § 4.2.3.1 si il existe plusieurs meilleurs opérateurs et la prise en compte de la mémoire) :

$$\forall o_i \in O_{cand}^{(n)}, \quad \exists p_i^{best} / \quad \sigma^{(n)}(o_i) = \min_{p_k \in Opr} \sigma(o_i, p_k)$$

    pour chaque opération, mémoriser :

    - ◇ son meilleur opérateur p_i^{best}
    - ◇ sa date de début obtenue sur ce meilleur opérateur : $S^{(n)}(o_i) = S(o_i)$ pour $\Pi(o_i) = p_i^{best}$
    - ◇ la valeur de la fonction de coût calculée pour cet opérateur : $\sigma^{(n)}(o_i)$

  - Pour maximiser l'utilisation des processeurs, construire la liste de candidats restreinte $O_{candI}^{(n)}$ qui correspond aux opérations de $O_{cand}^{(n)}$ dont la date de début au plus tôt est strictement inférieure à la date de fin au plus tôt de l'opération $o_{min}^{(n)}$ qui a la plus petite date de début au plus tôt de $O_{cand}^{(n)}$:

$$o_{min}^{(n)} = \{o_j \in O_{cand}^{(n)} \quad / \quad S^{(n)}(o_j) = \min_{o_i \in O_{cand}^{(n)}} S^{(n)}(o_i)\}$$

    La liste restreinte des candidats s'écrit alors :

$$\begin{aligned} O_{candI}^{(n)} = \quad & \{o_i \in O_{cand}^{(n)} \quad / \quad S^{(n)}(o_i) < E^{(n)}(o_{min}^{(n)}) \\ & avec \quad E^{(n)}(o_{min}^{(n)}) = S^{(n)}(o_{min}^{(n)}) + \Delta(o_{min}^{(n)}, p_{o_{min}^{(n)}}^{best})\} \end{aligned}$$

  - Parmi cette liste restreinte, il faut

    - ◇ sélectionner l'opération $o_{elue}^{(n)}$ la plus urgente à ordonnancer : (Cf. §4.2.3.2 pour le cas où plusieurs opérations ont la même urgence maximale)

$$o_{elue}^{(n)} = \{o_j \quad / \quad \sigma_{best}^{(n)}(o_j) = \max_{o_j \in O_{candI}^{(n)}} \sigma^{(n)}(o_j)\}$$

    - ◇ puis l'ordonnancer sur l'opérateur qui avait été déterminé : tel que $\Pi(o_{elue}^{(n)}) = p_{o_{elue}^{(n)}}^{best}$, tout en construisant les communications des dépendances de données entre $o_{elue}^{(n)}$ et chacun de ses prédécesseurs $\Gamma^{-1}(o_{elue}^{(n)})$ ce qui fera l'objet du § 4.2.3.5.

◇ cette opération est ajoutée à l'ensemble des opérations ordonnancées de la prochaine itération : $O_{ordo}^{(n+1)} = O_{ordo}^{(n)} \cup o_{elue}^{(n)}$

∘ Mettre à jour la liste pour la prochaine itération :

◇ ajouter à la liste, les successeurs de $o_{elue}^{(n)}$ qui sont devenus ordonnancables $\Gamma_{ordo}^{(n+1)}(o_{elue}^{(n)})$ (tous leurs prédécesseurs doivent être ordonnancés, sauf les prédécesseurs de type retard et constante, Cf. § 4.2.3.3 et § 4.2.3.4) :

$$\Gamma_{ordo}^{(n+1)}(o_{elue}^{(n)}) = \Gamma(o_{elue}^{(n)}) \setminus \{o_j \in \Gamma(o_{elue}^{(n)}) / \Pi(\Gamma^{-1}(o_j)) \neq \emptyset\}$$

◇ retirer de la liste l'opération qui a été ordonnancée lors de cette itération :

$$O_{cand}^{(n+1)} = O_{cand}^{(n)} \setminus \{o_{elue}^{(n)}\}$$

• Ajout des opérations de type constante (Cf. § 4.2.3.4).

Remarque 33 *Cet algorithme n'effectue pas les optimisations spécifiques aux opérations conditionnées s'exécutant exclusivement les une des autres. Cette optimisation a été traitée dans [102], l'algorithme obtenu peut être transposé directement à l'algorithme présenté ci-dessus car le modèle d'architecture et les optimisations réalisées ici sont toujours valables dans le cas conditionné.*

4.2.2.2 Fonction de coût

Notre fonction de coût, appelée pression d'ordonnancement et notée σ, mesure le degré d'urgence à ordonnancer une opération. Pour cela elle tient compte de la flexibilité (Cf. § 4.1.5) d'une opération (plus une opération a de la flexibilité, moins il est urgent de l'ordonnancer), mais aussi de l'allongement du chemin critique qu'induit l'ordonnancement d'une opération o_i. Ce dernier est décrit par $P^{(n)}(o_i, p_j)$, la pénalité d'ordonnancement engendrée par l'opération o_i lorsqu'elle est ordonnancée sur l'opérateur p_j. Si on note $\mathcal{R}_{ij}^{(n)}$ la longueur du chemin critique à l'étape n où l'on a ordonnancé o_i sur p_j, et $\mathcal{R}^{(n-1)}$ la longueur du chemin critique à l'étape précédente, la pénalité d'ordonnancement s'écrit :

$$P^{(n)}(o_i, p_j) = \mathcal{R}_{ij}^{(n)} - \mathcal{R}^{(n-1)}$$

La flexibilité d'ordonnancement (Cf. § 4.1.5) engendrée par l'opération o_i quand elle est ordonnancée sur p_j est donnée par :

$$F^{(n)}(o_i, p_j) = \mathcal{R}^{(n)} - \bar{S}^{(n)}(o_i, p_j) - S^{(n)}(o_i, p_j)$$

La pression d'ordonnancement est construite à partir de la différence de ces deux fonctions :

$$\sigma^{(n)}(o_i, p_j) = P^{(n)}(o_i, p_j) - F^{(n)}(o_i, p_j)$$

La pénalité d'ordonnancement d'une opération est donc nulle tant que le chemin critique ne s'allonge pas, c'est à dire tant que la date de début d'exécution n'a pas dépassé la date de début au plus tard. Dans ce cas, la valeur de la pression d'ordonnancement est égale, au signe près, à sa flexibilité d'ordonnancement.

Lorsque la flexibilité d'ordonnancement devient nulle, l'opération est dite critique car elle fait augmenter la longueur du chemin critique (croissance de la pénalité). La pression d'ordonnancement est alors égale à la pénalité d'ordonnancement.

Cette fonction de coût est satisfaisante pour évaluer l'urgence à ordonnancer une opération car, plus la pression d'ordonnancement est élevée, plus il est important d'ordonnancer cette opération étant donné l'allongement du chemin critique qu'elle induit.

4.2.3 Améliorations de l'heuristique

4.2.3.1 Cas où plusieurs opérateurs induisent le même coût

Pour rechercher le meilleur opérateur de chaque opération candidate, on l'ordonnance sur chaque opérateur capable de l'exécuter, on ne conserve alors que l'opérateur qui minimise la fonction de coût. Dans certain cas il est possible que plusieurs opérateurs induisent le même coût minimal, il faut donc les départager. Pour cela il existe différentes techniques, parmi lesquelles, le "choix aléatoire" ou les méthodes de voisinage locale sont les plus répandues. Un algorithme de retour-arrière ("back-tracking") à été proposé dans [102]. Il est basé sur la construction et destruction d'autant de graphes d'implantations qu'il y a de choix différents possibles. A l'issue ce cet algorithme, le choix qui minimise la longueur du chemin critique est retenu. Si cet méthode complète bien les heuristiques gloutonnes, elle présente l'inconvénient de requérir une durée d'exécution d'autant plus grande qu'il y a de cas où il existe à nouveau des opérations équivalentes dans les étapes suivantes de l'heuristique. Pour chacune de ces étapes il faut alors décider si l'on explore toute les solutions ou si l'on fait un choix aléatoire avec les conséquences que l'on connaît.

Nous proposons ici une solution complémentaire qui ne se substitue pas aux méthodes de voisinages mais qui améliore sensiblement l'heuristique. Ainsi nous proposons de tenir compte de l'occupation des mémoires de chaque opérateur pour sélectionner l'opérateur. Parmi les meilleurs opérateurs d'une opération, nous sélectionnerons celui qui est connecté aux mémoires programmes et données possédant le plus grand espace libre. Ceci permet de bien répartir les opérations sur les opérateurs et d'éviter de saturer rapidement la mémoire des opérateurs.

4.2.3.2 Cas où plusieurs opérations candidates induisent le même coût

Pour sélectionner l'opération à ordonnancer, nous comparons les valeurs des pressions d'ordonnancement de chaque opération de la liste restreinte de candidats de façon à choisir celle qui possède la plus grande pression d'ordonnancement. Si lors d'une étape de l'heuristique, plusieurs opérations possèdent la même pression d'ordonnancement, on dit qu'elle sont *´equivalentes* car la fonction de coût ne permet plus de les départager. Comme dans le cas précédent, il est possible d'appliquer une méthode de voisinage local ou de tirer un opérateur au hasard.

Nous proposons ici une solution complémentaire qui, comme précédemment, ne se substitue pas aux méthodes de voisinages mais qui améliore sensiblement l'heuristique. Dans le cas où plusieurs opérations sont équivalentes, nous proposons d'ordonnancer l'opération qui a la plus grande contrainte de distribution. En effet, dans le cas des architecture hétérogènes, toutes les opérations ne sont pas exécutable par tous les opérateurs. En sélectionnant celle dont le nombre d'opérateurs capable de l'exécuter est le plus faible on garantit que cette opération sera ordonnancée sur l'opérateur le plus adapté (celui qui lui induit la plus faible pression d'ordonnancement). Les autres opérations ayant une contrainte de distribution plus faible, les chances sont plus grandes de leurs trouver un opérateur adapté. Cependant, dans le cas où il existe plusieurs opérations équivalentes et dont les contraintes d'ordonnancement sont elles aussi identiques, le choix aléatoire ou les méthodes de voisinages peuvent être utilisés.

4.2.3.3 Sommet retard

Les sommets retard (ensemble O'_{alloc}) ont un rôle particulier puisqu'ils permettent d'implanter les dépendances de données inter-itération de l'algorithme (Cf. § 2.2.2, p. 56). Pour cela, chaque retard est implanté par une opération qui recopie le contenu du sommet allocation, associé à la dépendance de données qui connectent le retard à ses prédécesseurs, dans le sommet allocation associé à la dépendance de données qui le connectent à ses successeurs. Cette recopie ne doit être effectuée que lorsque tous les successeurs mais aussi tous les prédécesseurs ont utilisés les données contenues dans les sommets allocation. Les retards ne

sont donc ordonnançables qu'à partir du moment où tous leurs prédécesseurs ont été ordonnancés, ansi que tous leurs successeurs.

4.2.3.4 Sommet constante

Les sommets constantes (ensemble O'_{const}) correspondent à des opérations qui produisent des données identiques lors de chaque itération de l'algorithme de l'application (Cf. § 2.3, p. 57). Il suffit donc de les exécuter lors de la première itération. Ces opérations n'ont donc pas besoin d'être traitées lors de toutes les étapes de l'heuristique. Pour évaluer l'ordonnançabilité de leurs successeurs il faut considérer qu'elles ont été ordonnancées. La dernière étape de l'heuristique consiste à distribuer puis à ordonnancer ces opérations (ce sont les seules opérations qui ne sont pas distribuées et ordonnancées simultanément). Tout d'abord les opérations constantes sont distribuées sur chaque opérateur exécutant une opération consommatrice de leurs données, au besoin en les dupliquants de façon à éviter de communiquer des constantes. Enfin, comme ces opérations n'ont pas de prédécesseur, nous les ordonnançons de façon à ce qu'elles soient les premières opérations exécutées.

4.2.3.5 Construction des communications

Dans cette section nous nous intéressons à l'implantation de chaque dépendance de données d_{ji} entre l'opération o_i que l'on cherche à ordonnancer, et chacun de ses prédécesseurs $o_p \in \Gamma^{-1}(o_i)$ ordonnancés lors des itérations précédentes. Nous choisissons d'implanter les dépendances dans l'ordre croissant des dates de fin des opérations productrices. Ainsi, il est possible de commencer et donc de terminer les communications au plus tôt ce qui permet de débuter l'exécution de l'opération o_i le plus tôt possible.

Construction des tables de routage Avant de débuter l'heuristique et pour gérer la construction des communications, il faut construire des routes entre chaque couple d'opérateur du graphe d'architecture. Pour cela nous construisons des *tables de routages* pour chacun des opérateurs du graphe de l'architecture. Ces tables permettent d'encoder le résultat du routage présenté dans les paragraphes 3.1.2 et 3.2.2. A partir de ces tables, il sera possible d'implanter les dépendances de données entre n'importe quel couple d'opérateurs. Lors de la construction de ces tables, comme nous voulons que l'opération consommatrice o_i commence au plus tôt, nous ne conservons que les routes les plus courtes (nombre de RAM, SAM et communicateurs intermédiaires) entre chaque couple d'opérateurs. Si il existe plusieurs routes parallèles de même longueur, nous les conservons toutes, de façon à pouvoir sélectionner celle qui permettra de terminer la communication au plus tôt lors de l'implantation de chaque dépendance de données (Cf. § 4.2.3.5). Nous verrons également que cela permet de router plusieurs données en parallèles.

Remarque 34 *Il existe d'autres critères de sélection de routes à expérimenter, ainsi nous envisageons de faire intervenir la bande passante moyenne des routes et non plus seulement leur longueur. Le classement selon la longueur a cependant l'intérêt de minimiser le surcoût d'espace mémoire intermédiaire nécessaire au stockage des données (puisque l'on minimise alors le nombre de sommets allocation).*

Pour chaque opérateur p_i, sa table de routage indique pour chaque autre opérateur $p_j \neq p_i$, la distance qui les sépare, ainsi que l'ensemble des communicateurs et RAM connectés à p_i qui permettent de les joindre. Étant donné que nous ne construisons que les routes les plus courtes entre chaque couple d'opérateurs, l'algorithme qui remplit les tables de routage est relativement simple :

- Initialisation de la distance d à 1

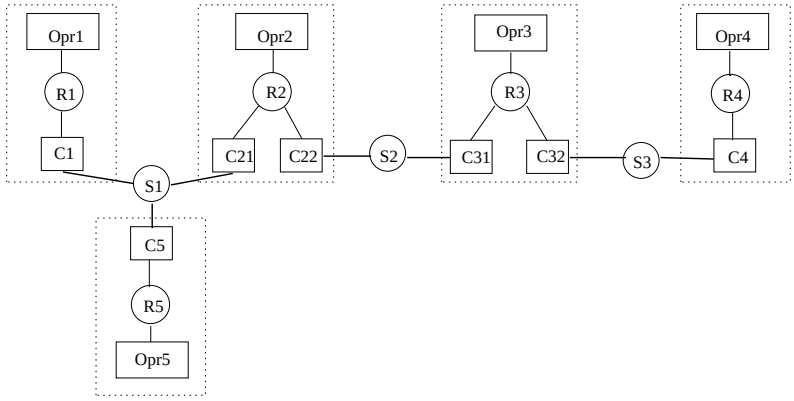


FIG. 4.8: *Graphe d'architectures*

Opr1	C1	R1	Dist
Opr1	-	•	1
Opr2	•	•	3
Opr3	•	•	5
Opr4	•	•	5
Opr5	•	•	3

...

Opr2	C21	C22	R2	Dist
Opr1	-	•	•	3
Opr2	-	•	•	1
Opr3	-	•	•	3
Opr4	-	•	•	5
Opr5	•	-	•	3

etc ..

FIG. 4.9: *Exemples de tables de routage (les points marquent la possibilité d'atteindre l'opérateur (ligne) par le média (colonne)*

- Tant que pour chaque opérateur, il manque une route pour atteindre un opérateur (c'est à dire tant que la table de routage d'un opérateur est incomplète) :
  - pour chaque opérateur de l'architecture :
    - ◇ recherche des opérateurs distant de d routes élémentaires, pour chaque nouvel opérateur atteint, mémoriser dans la table de routage de l'opérateur courant et de l'opérateur atteint :
      - ⋆ le communicateur utilisé pour atteindre le nouvel opérateur,
      - ⋆ la distance d
- Fin, toute les tables de routages sont remplies

Exemple 4.2.1 *Voici les tables de routages (fig 4.9) des opérateurs $Opr1$ et $Opr2$ du graphe d'architecture de la figure 4.8. La première table indique bien que toutes les communications inter-opérateurs doivent passer par la RAM $R1$ et le communicateur $C1$. La table de routage de l'opérateur montre que les communications à destination de $Opr1$ et $Opr5$ doivent passer par le communicateur $C21$.*

Implantation sur route iso-opérateur Lorsque l'opération émettrice d'une dépendance de données d_j est ordonnancée sur le même opérateur que l'opération o_i ($\Pi(\gamma^{-1}(d_j)) = \Pi(o_i)$), nous avons vu (§ 3.2.3.2) que d_i est implantée par un sommet allocation associé à une RAM et d'autant de sommets identité qu'il y a de sommets Bus/Mux/Demux entre l'opérateur et la RAM choisie. Parmi l'ensemble des RAM connectées à l'opérateur, le sommet allocation sera distribué, de préférence, sur une RAM non partagée de façon à

conserver autant d'espace mémoire que possible dans les RAM partagées. En effet, les RAM partagées doivent être "économisées" pour réaliser les communications inter-opérateurs.

Parmi l'ensemble des RAM non partagées connectées, on choisira le sommet connecté par la route de plus grande bande passante possédant encore suffisamment d'espace mémoire, et sur laquelle des sommets allocation ont déjà été distribués. Cette règle garantit que chaque mémoire est bien remplie.

Implantation sur route à mémoire RAM sans communicateur Lorsque l'opération productrice et l'opération consommatrice sont distribuées sur des opérateurs connectés par une ou plusieurs RAM, nous avons vu que la dépendance de données était implantée par un sommet allocation distribué sur l'un des sommets RAM. Comme précédemment, si il existe plusieurs RAM, il faut choisir celui qui appartient à la route de plus grande bande passante, possédant encore suffisamment d'espace, et sur lequel des sommets allocation ont déjà été distribués.

Implantation sur routes composées L'implantation d'une dépendance de données d_j sur une route s'effectue en autant d'étapes qu'il y a d'opérateurs sur cette route. Notons $p^{(m)}$ l'opérateur "courant" de l'étape m, o_{prod} l'opération productrice de d_j et o_i l'opération consommatrice de d_j qui vient d'être ordonnancée sur un opérateur. On a donc $\gamma^{-1}(d_i) = o_{prod}$ et $o_i \in \gamma(d_i)$.

Initialement, l'opération productrice et l'opération consommatrice sont toutes deux distribuées sur des opérateurs différents et sont connectées par la dépendance de données d_i. Pour implanter cette dépendance de données sur la route qui joint leurs opérateurs respectifs nous utilisons l'algorithme suivant :

Algorithme simplifié

- l'opérateur producteur est l'opérateur courant, L est la distance qui sépare l'opérateur producteur de l'opérateur consommateur (celui qui exécute l'opération consommatrice),
- tant que l'opérateur courant n'est pas l'opérateur consommateur :
  - si la donnée à communiquer diffuse, rechercher une opération de communication existante qui permet d'atteindre un opérateur p qui soit à une distance inférieure à L de l'opérateur consommateur. Si une telle opération de communication est trouvée, elle est utilisée, l'opérateur courant devient l'opérateur p,
  - sinon (pas de diffusion ou pas d'opération de communication trouvée) créer et ordonnancer une opération de communication sur le communicateur qui permet d'atteindre un opérateur p qui soit à une distance inférieure à L de l'opérateur consommateur (ce communicateur est déterminé grâce aux tables de routages). Si plusieurs communicateurs sont possibles, on recherche celui qui permet d'achever la communication au plus tôt. L'opérateur courant devient l'opérateur p.
- Fin (l'opérateur courant est l'opérateur consommateur).

(L'algorithme complet est donné sur la page suivante)

Algorithme détaillé

- On part de l'opérateur producteur : $p^{(m)} = \Pi(o_{prod})$, soit L la distance (tirée de la table de routage) qui le sépare de $\Pi(o_i)$,
- Tant que l'opérateur courant n'est pas l'opérateur de l'opération consommatrice de d_i (tant que $p^{(m)} \neq \Pi(o_i)$) faire :
  - pour chaque communicateur c_{emet} qui permet d'atteindre $\Pi(o_{prod})$ selon la table de routage de $p^{(m)}$ faire :
    - Si d_i possède plusieurs consommateurs (diffusion), on cherche à *réutiliser une communication* éventuelle : si il existe déjà une opération de communication émettrice (SEND ou RECEIVE) correspondant à l'implantation de d_i :
      - rechercher l'opération consommatrice (RECEIVE ou read) correspondante et le communicateur c_{recep} sur lequel elle est ordonnancée,
        - si l'opérateur p connecté à c_{recep} est $\Pi(o_i)$ ou si la table de routage de cet opérateur indique que p est à une distance de $\Pi(o_i)$ inférieure à L, la communication est réutilisée, il n'y a pas de sommet à ajouter dans cette étape,
        - si l'opération émettrice est un write et qu'il existe un communicateur c_{recep} connecté à un opérateur p tel que $p = \Pi(o_i)$ ou tel que la table de routage de p indique qu'il est à une distance inférieure à L de $\Pi(o_i)$, il faut ajouter une opération de communication READ sur c_{recep} afin de lire les données déjà dans la RAM partagée, nous ajoutons une précédence entre le premier READ et celui que l'on vient d'ajouter de manière à ce que les durée calculées lors de cette précédente communication ne soient pas modifiées par l'arbitrage de la RAM.
  - Si pas de réutilisation, il faut construire une communication :
    1. recherche de la meilleure route par construction/destruction :
       - pour chaque communicateur c_{emet} qui permet d'atteindre $\Pi(o_{prod})$ selon la table de routage de $p^{(m)}$ faire :
         - ajouter un sommet allocation et des sommets identité éventuels sur la route qui connecte $p^{(m)}$ et c_{emet},
         - ajouter une opération de communication émettrice (SEND ou WRITE) sur c_{emet},
         - ajouter un sommet allocation sur la RAM ou la SAM inter-communicateurs m_{par} connectée à c_{emet},
         - parmi les communicateurs connectés à m_{par} sélectionner le communicateur c_{recep} connecté à un opérateur p tel que $p = \Pi(o_i)$ ou tel que la table de routage de p indique qu'il est à une distance inférieure à L de $\Pi(o_i)$
         - ajouter une opération de communication réceptrice (RECEIVE ou READ)
       - mesurer la date de fin de la communication (date de fin de l'opération réceptrice) exécutée par c_{recep},
       - détruire les sommets ajoutés,
    2. construire la communication en partant du communicateur c_{emet} qui, parmi ceux explorés, a induit la plus petite date de fin, soit p l'opérateur atteint par cette communication,
  - $p^{(m+1)} = p$ pour la prochaine itération

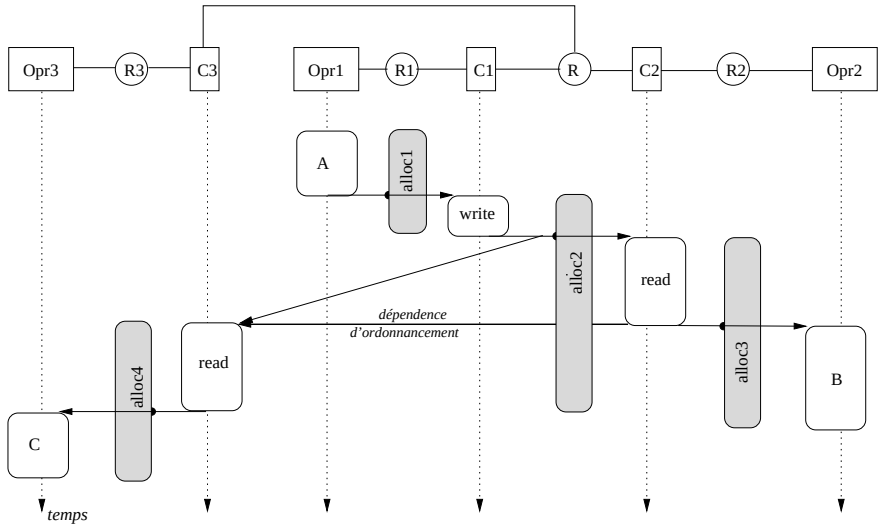


FIG. 4.10: *Construction des communications*

Exemple 4.2.2 *Dans l'exemple de diagramme temporel d'implantation de la figure 4.10, lors de l'ordonnancement de l'opération C sur $Opr3$, il y a réutilisation de la communication de la dépendance de données entre les opérateurs $Opr1$ et $Opr2$.*

4.3 Optimisation de la mémoire

L'heuristique d'optimisation a été étendue afin de prendre en compte les mémoires introduites dans le nouveau modèle d'architecture. Pour optimiser l'utilisation des mémoires de données nous effectuons de la ré-allocation statique de mémoire. La ré-allocation et plus généralement le "ramassage de miettes" (garbage collect) est souvent traitée dynamiquement par les langages et outils de haut niveau (orientés objets) tel que Java[20] ou Smalltalk[13]. Périodiquement, à des instants plus ou moins précis, le ramasse miette désalloue les espaces mémoires qui ne sont plus utilisés (dès lors qu'ils ne sont plus référencés) et réorganise la mémoire (défragmentation) de façon à ne plus laisser de petits espaces mémoire inutilisables entre les espaces mémoire alloués. L'espace libéré peut être ré-alloué aux besoins ultérieurs de l'algorithme. Cependant, étant donné le surcoût temporel qu'induit l'exécution d'un ramasse miettes, ils sont rarement utilisés dans les applications temps réel embarquées, ils sont en effet beaucoup plus adaptés aux systèmes de type stations de travail multi-utilisateurs afin de libérer périodiquement de la mémoire pour d'autres programmes susceptibles de s'exécuter sur la même machine.

Dans le cadre de la méthodologie AAA, où la distribution et l'ordonnancement sont effectués hors-ligne par des heuristiques, nous possédons toutes les informations nécessaires pour effectuer de la ré-allocation statique de la mémoire. En effet, dans le paragraphe 4.1.6.2, pour chaque sommet allocation nous avons défini un intervalle d'utilisation minimal (durée de vie) borné par les dates de leur opération productrice et

de leur(s) opération(s) consommatrice(s). Après la construction du graphe d'implantation par l'heuristique, nous pouvons calculer les bornes de tous les sommets allocation du graphe. Si, au sein d'une même RAM, l'intersection des durées de vie de certains sommets est nulle, et que ces sommets modélisent une quantité de données équivalente, il est possible d'utiliser le même espace mémoire pour chacun de ces sommets. Au niveau du graphe d'implantation, les sommets allocation pour lesquels on veut réutiliser un espace mémoire déjà alloué, sont remplacés par un nouveau type de sommets nommé *alias*. Soit O''_{alias} le sous-ensemble de ces sommets, ils sont étiquetés par le noms des sommets allocation dont ils réutilisent l'espace alloué.

$$\begin{array}{ll} \forall a''_1, a''_2 \in O''_{alloc} & / \quad q'(\alpha^{-1}(a''_1)) = q'(\alpha^{-1}(a''_2)), \\ & si \quad S(a''_1) \geq E(a''_2), \quad a''_1 \quad devient \quad b''_1 \quad avec \quad b''_1 \in O''_{alias} \\ & sinon, \quad si \quad S(a''_2) \geq E(a''_1), \quad a''_2 \quad devient \quad b''_2 \quad avec \quad b''_2 \in O''_{alias} \end{array}$$

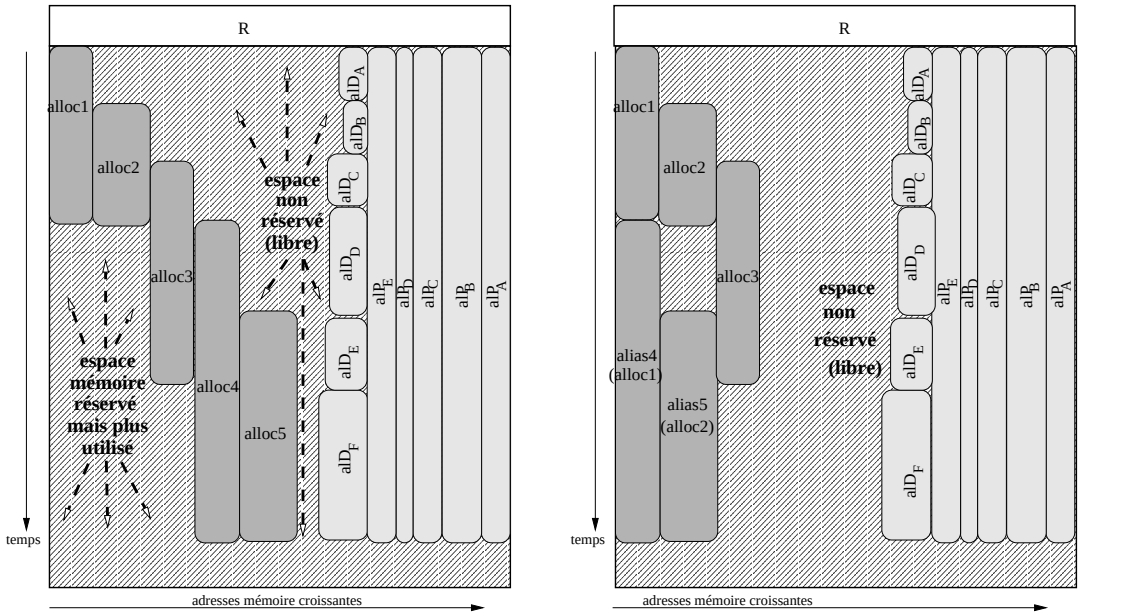
Lors de la génération d'exécutif (§ 7.2.1.3, p. 154), tous les noms d'alias seront substitués par le nom des sommets allocation correspondant. Cette optimisation de la mémoire est modélisée par la relation $\mathcal{R}_{Realloc}$:

$$\begin{array}{rcl} (G_{ordoR}, G'_{ar}) & \xrightarrow{\mathcal{R}_{Realloc}} & (G_{reallocR}, G'_{ar})(.) \\ (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} & \xrightarrow{\mathcal{R}_{Realloc}} & (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \\ \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}}, & & \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}}, \\ H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}) & & H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}) \end{array}$$

Comme cette relation ne fait que transformer certains sommets allocation en sommets alias sans modifier les dépendances de données entre les opérations, l'ordre partiel du graphe d'algorithme est inchangé et toujours décrit par :

$$\trianglelefteq = (\bigcup_{p \in s_{opr}} \prec_p) \cup (\bigcup_{c \in S_{com}} \prec_c) \cup (\bigcup_{p \in S_{opr}, c \in S_{com}} \preceq_c pc)$$

Exemple 4.3.1 *La partie a) de la figure 4.11 présente le diagramme temporel d'utilisation d'une mémoire R avant optimisation par ré-allocation. La partie b) de cette figure présente le diagramme temporel optimisé de cette mémoire. La hauteur de chaque sommet allocation correspond à leur durée de vie, et leur largeur à la quantité de données allouée. Les sommets* `alloc4` *et* `alloc5` *réservent la même quantité de données que les sommets* `alloc1` *et* `alloc2` *et il n'y a pas intersection entre leurs durées de vie respectives. Il est donc possible de remplacer* `alloc4` *et* `alloc5` *par des sommets* `alias4` *et* `alias5` *chacun étiqueté par le nom des sommets allocation* `alloc1` *et* `alloc2` *dont il réutilise l'espace mémoire réservé.*



a) Diagramme temporel d'utilisation de la RAM R

b) Diagramme temporel d'utilisation optimisée de la RAM R

FIG. 4.11: *Ré-allocation mémoire*

Deuxième partie

Génération automatique d’exécutifs distribués

Chapitre 5

État de l'art

Sommaire

5.1 Exécutif standard

Dans notre méthodologie, l'implantation de l'algorithme (décrit par le graphe d'algorithme) d'une application sur un calculateur monoprocesseur, consiste à traduire cet algorithme en un programme, chargé puis exécuté par l'unique processeur. Le programme correspond alors à un ensemble d'instructions exécutées séquentiellement par le séquenceur d'instruction du processeur.

Quand le calculateur est une machine multiprocesseur dont les séquenceurs d'instructions sont capables de fonctionner en parallèle, l'implantation parallèle de l'algorithme consiste à le traduire en un ensemble de programmes chargés puis exécutés simultanément sur chaque processeur de l'architecture (un programme par processeur). Ces programmes coopèrent (communiquent) pour réaliser les fonctionnalités de l'algorithme. L'ensemble de ces programmes forme le logiciel de l'application.

L'exécutif est la partie du logiciel qui gère les ressources matérielles pour les allouer aux besoins de l'algorithme, ou à des besoins annexes (par exemple le chronométrage ou la génération de traces d'exécution lors de la phase de mise au point).

Les ressources à allouer sur une architecture multi-composants vue au niveau macroscopique sont :

– les espaces mémoires connectés aux opérateurs, pour le code et pour les données,

– le temps des opérateurs, plus précisément celui de leur séquenceur d'instructions,

– le temps des communicateurs (séquenceurs de transferts) qui partagent les mémoires.

Les exécutifs traditionnels, appelés aussi *systèmes d'exploitation* (abrégé OS pour Operating System en anglais), sont conçus séparément des applications qu'ils supportent. Ils fournissent donc une vaste gamme de services pour tenter de répondre aux besoins variés des concepteurs d'applications, qui utilisent des méthodes variées de conception et d'implantation de leurs algorithmes. Parmi ces services on trouve généralement :

- la communication,
- la synchronisation,

- la gestion et l'ordonnancement des tâches (selon des priorités associées à chacune d'elles, l'exécutif fournit souvent plusieurs politiques d'ordonnancement),
- la gestion de la mémoire (avec des mécanismes de protection plus ou moins élaborés),
- la gestion des interruptions,
- la gestion des périphériques,
- la gestion des fichiers et des supports,
- le traitement des erreurs et des exceptions,
- la gestion du temps.

La plupart de ces OS ne garantissent pas une exécution déterministe des applications qu'ils exécutent, les appels systèmes (services) ne sont pas déterministes. Pourtant, dans le cadre d'applications temps réel embarquées il est indispensable que l'OS soit déterministe pour pouvoir garantir que l'exécution de l'application soit prévisible et respecte ainsi toujours toutes les contraintes (durée d'exécution, sécurité de fonctionnement . . .). Pour répondre à ces exigences il existe des OS Temps Réel (RTOS) qui garantissent une exécution déterministe des applications qu'ils implémentent. Comme les premiers, ils sont conçus séparément des applications qu'ils supportent et fournissent donc un large éventail de services et surtout un ordonnanceur basé sur un mécanisme de priorités liés aux contraintes temps réelles. L'application à implanter est décrite sous formes de tâches communiquantes possédant chacune une priorité fixe ou dynamique. A des instants périodiques, ou après la fin d'une tâche, ou après une interruption, l'ordonnanceur de l'OS décide de la tâche à exécuter en fonction des priorités. Dans le monde des OS, il existe un premier standard, POSIX[72] (développé par l'IEEE [1] et normalisé par l'ANSI [2] et ISO [3]), définissant un jeu d'appel système normalisé. POSIX est basé sur UNIX. La majorité des OS standards actuels sont compatibles POSIX. Il existe une extension temps réel à POSIX, nommée 1003.b, qui permet d'utiliser l'OS pour des applications temps réel. Ces extensions traitent de la gestion du temps et de la priorité des processus ainsi que des appels systèmes spécifiques à la communication inter-processus. Dans le cadre des systèmes embarqués automobile, des constructeurs automobiles (BMW, Daimler-Benz/Mercedes, Opel, Volkswagen, PSA, Renault) et des équipementiers (Bosch, Siemens) se sont regroupés pour proposer une standardisation (OSEK/VDX) des systèmes d'exploitation automobile afin d'assurer la portabilité du code de leurs applications et réduire le coût des développements.

Il existe de nombreux RTOS dont les plus répandus sont VxWorks, Chorus, LynxOS, QNX, Virtuoso, RTAI, RTLinux. VxWork développé par Windriver, est le plus utilisé aujourd'hui, avec 12% du marché. VxWork, comme RTAI ou RTLinux sont basés sur un noyau monolithique relativement gros et difficile à étendre, donc difficile à maintenir, caractériser et faire évoluer. Ils possèdent un mécanisme classique de protection de la mémoire pour que les tâches qui s'exécutent n'écrivent pas dans l'espace des autres processus, cependant le noyau accède à quasiment tout l'espace mémoire avec la complexité de mise au point que cela entraîne. QNX est en revanche basé sur un très petit noyau, baptisé en conséquence "Neutrino", conçu de façon à être facilement extensible (un noyau avec support TCP/IP, application de type navigateur internet etc, tient sur moins d'1Mo). Il dispose d'un mécanisme de protection mémoire beaucoup plus évolué qui limite les interactions entre les processus, même ceux du noyau. Très en vogue actuellement, il est bien adapté aux petites applications embarquées et envahit maintenant le segment de plus grosses applications. En ce qui concerne les architectures multiprocesseurs, Chorus, LynxOS et QNX sont des OS distribués qui

1. Institute of Electrical and Electronics Engineers, http://www.ieee.org/
2. American National Standards Institue, http://www.ansi.org/
3. ISO/IEC ISP 15287-2:2000, International Organization for Standardization, http://www.iso.ch/

offrent des services de communications puissant (FIFO, boite aux lettres, estampillages), leur structure est proche de la norme POSIX. Virtuoso a, dès son origine, été conçu pour des applications de calcul intensif (traitement du signal) fonctionnant sur architecture multiprocesseur. Cet outil prend totalement en charge la gestion des aspects distribués de l'application grâce à un concept de processeur virtuel unique ("Virtual Single Processor", VSP). Le concepteur doit seulement attribuer un processeur à chaque tâche de son algorithme (partitionnement), sans se préoccuper ni de l'allocation des mémoires, ni des communications qui sont automatiquement prises en charge par Virtuoso. Cependant il ne fournit pas d'outils de placement automatique, tout au plus des traces d'exécution en-ligne ou hors-ligne (il est comparable en ce sens aux logiciels GEDAE[62] et Trapper[91] qui seront présentés dans la troisième partie). RTAI et RT-LINUX sont basés sur les noyaux de Linux, modifiés de façons à garantir le temps réel. En fait ils renferment un mini ordonnanceur qui considère les tâches du noyau linux comme des tâches standards, c'est à dire qu'elles possèdent une priorité qui n'est pas la plus élevée. Ces tâches, qui ne sont plus les plus importantes (vis à vis des tâches temps réel) ont donc été rendues interruptibles pour pouvoir garantir des délais d'exécution quelque soit la tache en cours d'exécution. Avec l'arrivée de ces RTOS spécialisés, les OS généralistes sont maintenant qualifiés de GPOS (*General Purpose OS*) parmi lesquels sont classés par exemple Linux, Solaris, Windows 9x, NT, CE.

5.2 Exécutif "sur mesure"

Certains concepteurs ne veulent consacrer qu'un minimum de leur temps à l'implantation de leur algorithme, aussi attendent-ils de l'exécutif qu'il leur fournisse des services d'allocation automatique des ressources, au prix de surcoûts non seulement d'allocation dynamique (migration de processus pour équilibrage de charge des processeurs, ramasse-miettes de la mémoire (garbage collector), multitâches en temps partagé ou "time-slicing", paquetage et routage des communications ...) mais aussi auto-protection de l'exécutif contre d'éventuelles erreurs de programmation (vérification de la cohérence des requêtes d'allocation, procédures d'exception en cas d'incohérence ou d'insuffisance de ressource disponible, chiens de garde de détection de perte de messages ou d'interblocage) ainsi que divers outils d'analyse en ligne ou hors-ligne. Ce premier type de conception vise le prototypage rapide, où les objectifs prioritaires consistent à démontrer rapidement la faisabilité de l'application et éventuellement un premier dimensionnement de l'architecture matérielle et de la complexité logicielle pour estimer le coût d'industrialisation.

D'autres concepteurs veulent tirer parti au maximum des performances de l'architecture, aussi ont-ils besoin de contrôles bas niveau de l'allocation de l'architecture (possibilité d'écrire des programmes d'interruption à faible temps de réponse, contrôle du gestionnaire mémoire et du cache pour accélérer les temps d'accès, contrôle fin de la politique d'ordonnancement du gestionnaire multitâche ...) pour réduire les surcoûts cités ci-dessus, mais au prix d'un temps de mise au point plus important. Comme tous ces problèmes sont spécifiques à chaque application, ces concepteurs doivent construire des exécutifs "sur mesure" pour leurs applications. Ce deuxième type de conception vise l'industrialisation, où les objectifs prioritaires sont de réaliser une application (logiciel et matérielle) qui respecte toujours des contraintes temps réel, un certain degré de sûreté de fonctionnement, en respectant aussi les critères d'embarquabilité et souvent en essayant de coûter le moins cher possible. Ces deux derniers points conduisent naturellement à la minimisation de l'architecture dont l'utilisation doit être fortement optimisée.

Pour satisfaire les contraintes apparemment contradictoires de ces deux types de conceptions visant des objectifs différents, et ainsi factoriser les travaux de développement des applications temps réel distribuées en vue de réduire les cycles de développement, nous proposons de générer automatiquement les exécutifs distribués taillés sur mesure pour chaque application. En effet, d'une part l'automatisation de la génération de l'exécutif correspond bien aux besoins de développer rapidement des prototypes, et d'autre part les exécutifs produits étant taillés sur mesure, ils peuvent respecter les contraintes de minimisation liées à l'in-

dustrialisation. Nous présentons maintenant en détail l'ensemble des critères que doit satisfaire la génération automatique d'exécutifs, ainsi que les solutions que nous avons développés :

- l'exécution doit être déterministe car nous traitons des applications temps réel,
- dans le cadre du prototypage rapide, pour faciliter le portage d'un algorithme sur différentes architectures, il est souhaitable de pouvoir spécifier l'algorithme de façon générique (indépendemment du langage cible),
- dans le cadre du prototypage rapide, pour supporter facilement de nouvelles architectures sans remettre en cause le processus de génération automatique d'exécutif, il est souhaitable d'avoir une approche générique de l'allocation des ressources de l'architecture,
- les systèmes embarqués imposent de minimiser les ressources (minimisation des surcoûts d'exécution, optimisation de l'usage des mémoires etc).

Le déterminisme de l'exécution est basé sur un ordonnancement hors ligne non préemptif. Bien que ce type de politique d'ordonnancement restreigne le champ d'application de la méthodologie AAA, le nombre d'applications très critiques nécessitant cette approche justifie de s'y intéresser. Notons cependant que pour élargir le champs d'application de la méthodologie AAA, R. Kocik [56] s'est récemment attaché à introduire l'utilisation de la préemption dans cette méthodologie, tout en la minimisant et en étudiant ses conséquences. Pour que l'exécution soit déterministe, l'ordonnancement hors-ligne n'est pas suffisant, il faut en maîtriser tous les aspects. Les communications sont souvent considérées comme le point faible de l'ordonnancement statique car souvent gérées par des mécanismes dynamiques du système d'exploitation. Dans ce cas toutes les autres optimisations effectuées statiquement sont remises en cause par cet ordonnancement dynamique qui est plus complexe à maîtriser. Pour y remédier, il faut effectuer un ordonnancement statique des communications sans mécanismes dynamiques dûs au système d'exploitation. C'est sur cette hypothèse que les optimisations de l'implantation ont été réalisées et c'est sur ces principes que seront fondés les exécutifs que nous allons décrire.

Prototypage rapide : généricité de spécification algorithme Une spécification d'algorithme applicatif indépendante du langage cible de chaque processeur facilite l'utilisation d'architectures hétérogènes. C'est pourquoi nous avons choisi d'utiliser un macro-langage intermédiaire indépendant des architectures cibles qui utilise des bibliothèques de macros, qui elles sont dépendantes des architectures cibles. Le portage d'un algorithme sur différentes architectures consiste alors à porter ces bibliothèques sur les nouveaux types de composant de l'architecture, sans remettre en cause la spécification de l'algorithme ; seules les durées d'exécution caractérisant les opérations de l'algorithme doivent être modifiées.

Prototypage rapide : approche générique de l'allocation des ressources de l'architecture Le choix d'un ordonnancement statique nous permet en plus du déterminisme, d'avoir une approche basée sur un très petit nombre de services. Ces services correspondent à des macros d'allocation de ressources faisant parties de bibliothèques systèmes génériques que nous appelons noyaux génériques d'exécutifs. Bien que spécifique à chaque processeur, leur structure générique permet de construire automatiquement l'exécutif sur mesure pour l'application en allant chercher les macros systèmes nécessaires dans ces bibliothèques selon des règles que nous allons décrire. Le nombre de services étant restreint, la taille de chaque noyau est faible, il est donc possible de les porter facilement et rapidement sur n'importe quel type d'architecture sans

avoir à repenser le processus de génération en fonction des particularités de chaque processeur. Ce noyau est conceptuellement structuré en plusieurs familles de macros :

- macros de chargement arborescent des programmes,
- macros de gestion de la mémoire de données (sommet `alloc`),
- macros de contrôle structuré conditionnel et itératif,
- macros de lancement et de synchronisation des séquences,
- macros de transfert (sommets `read`, `write`, `SEND`, `RECEIVE`),
- macros de chronométrage.

Efficacité L'exécutif étant généré "sur mesure" pour l'application, il est très efficace. La faible complexité de chaque macro de l'exécutif, la maîtrise bas-niveau de l'architecture, permettent de faire un grand nombre d'optimisations qui ne seraient pas possibles sinon. En effet, dans la méthodologie que nous présentons dans cette thèse, un maximum de décisions et de vérifications d'allocations de ressources sont prises par l'heuristique d'optimisation, avant l'exécution. Ces décisions sont traduites automatiquement en un exécutif distribué (sans que l'utilisateur n'ait à l'écrire ni à le déboguer) qui, sur chaque processeur, séquence les opérations de calcul, alloue les espaces mémoires nécessaires au transfert des données entre opérations, et séquence les opérations de communications inter-processeurs. Le nombre nécessaire de ressources ayant été vérifié avant l'exécution par l'heuristique d'optimisation, l'exécutif n'a plus à effectuer les vérifications à l'exécution ni à se protéger contre d'éventuelles erreurs de programmation. Son surcoût, autant en espace mémoire qu'en temps d'exécution, est moindre que dans l'approche traditionnelle avec des RTOS.

Conclusion

Dans la méthodologie AAA l'exécutif constitue l'ossature sur laquelle s'appuie l'application. Il est généré sur mesure en fonction de l'algorithme et de l'architecture par assemblage de macros issues d'un noyau générique d'exécutif. Il est ensuite compilé et/ou lié avec les macro-opérations spécifiques à l'application pour produire un code binaire chargeable et exécutable directement sur l'architecture matérielle. Si l'usage d'un système d'exploitation résident est incontournable, il est encore possible d'utiliser ces techniques de génération automatiques d'exécutif qui fait alors appel à certains services de ce système d'exploitation. On comprend aisément que cette approche est moins performante bien qu'ayant l'intérêt de pouvoir utiliser du "logiciel sur étagère". Ces aspects ne sont pas abordés dans cette thèse.

Le temps que consacre traditionnellement l'utilisateur à écrire et à déboguer l'exécutif distribué de son application, peut être alors consacré à optimiser son implantation (en interagissant avec l'heuristique d'optimisation pour l'aider à trouver de meilleurs résultats si possible, ce qui peut l'amener à adapter la taille des grains de son algorithme, et à minimiser les ressources de son architecture).

Les trois chapitres suivants décrivent les trois phases (Cf. figure 5.1) qui permettent de générer l'exécutif conformément aux critères que nous venons de décrire :

- la première phase consiste à effectuer un certain nombre de transformations du graphe d'implantation produit par l'heuristique d'optimisation en ajoutant de nouveaux sommets qui correspondent aux opérations de contrôle des séquenceurs (synchronisations, itérations ...). Le graphe obtenu est appelé graphe d'exécution;

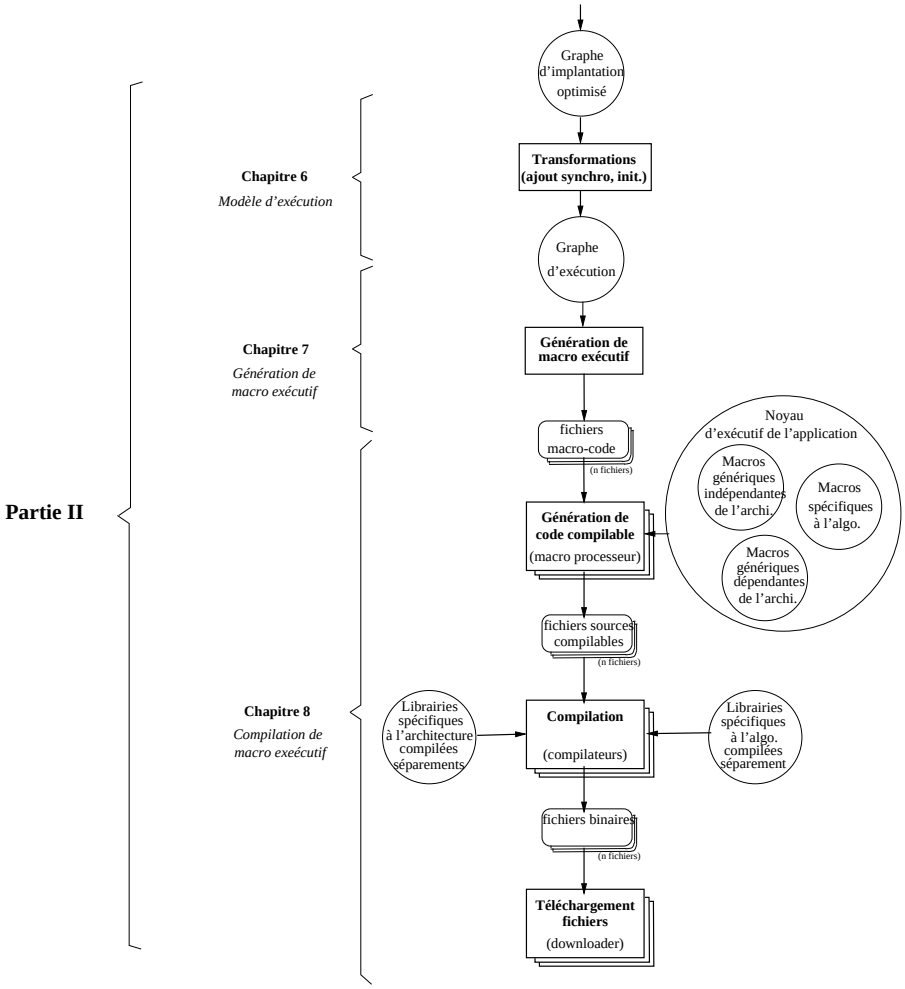


FIG. 5.1: *Plan de la quatrième partie*

- la seconde phase traduit le graphe d'exécution en un macro-exécutif intermédiaire, comprenant pour chaque processeur des appels de macros d'allocation mémoire, une séquence d'appels de macros de calcul et autant de séquences d'appels de macros de communication que de communicateurs accessibles au processeur. Ce macro-exécutif est générique, c'est-à-dire indépendant du langage cible préféré pour chaque type de processeur;
- la dernière phase traduit le macro-exécutif intermédiaire en exécutif compilable par le compilateur spécifique à chaque type de processeur cible. Cette traduction est basée sur un macro-processeur complété pour chaque type de processeur cible par un fichier de définitions de macros spécifiques à ce type de processeur. Nous appelons "noyau générique d'exécutif" le jeu de macros, extensible et portable entre différents langages cibles.

Chapitre 6

Modèle de macro-exécutif générique

Sommaire

Dans les chapitres 3 et 4 de la première partie nous avons défini comment, à partir d'un graphe d'algorithme et d'un graphe d'architecture, nous construisions un graphe d'implantation optimisé dans lequel l'ordre partiel d'exécution des opérations du graphe d'algorithme a été renforcé par l'insertion de précédences, chaque opération de l'algorithme a été ordonnancée sur un opérateur de l'architecture. Les dépendances de données initiales du graphe de l'algorithme ont été remplacées par des sous-graphes faits d'opérations de communications et d'allocations mémoire qui sont elles mêmes distribuées et ordonnancées sur les sommets du graphe de l'architecture.

Dans ce chapitre nous définissons la transformation du graphe d'implantation en graphe d'exécution synchronisé à partir duquel pourra être généré le macro-exécutif générique, qui sera étudiée dans le prochain chapitre.

Pour clarifier les explications qui vont suivre, ce chapitre est scindé en deux parties. Dans la première partie nous allons étudier les transformations à effectuer dans le cas d'un algorithme implanté sur le type d'architecture le plus simple, mono-opérateur. Dans la seconde partie nous étudierons les transformations supplémentaires nécessaires à l'implantation d'un algorithme sur une architecture multi-opérateur.

6.1 Monoprocesseur

6.1.1 Précédences

Comme nous l'avons montré plus haut, nous nous intéressons ici aux opérations associées aux séquenceurs d'instructions contenus dans les opérateurs et communicateurs puisque c'est pour eux qu'il faut générer l'exécutif. Par la suite, et conformément au modèle d'architecture présenté dans le chapitre 1, un

processeur correspond à un sous-graphe du graphe d'architecture constitué d'un unique opérateur et des communicateurs connectés à la même mémoire programme. Les opérations de calcul ou d'entrées écrivent les données qu'elles produisent dans des tampons mémoires (sommets allocation et alias distribués sur les mémoires) qui sont ensuite lus par d'autres opérations de calcul ou de sortie, selon les dépendances de données spécifiées dans le graphe d'algorithme. Étant donné qu'une opération est composée d'un ensemble d'instructions, dans le pire des cas il est possible que la dernière instruction de l'opération productrice soit une instruction d'écriture dans le tampon et que la première instruction de l'opération consommatrice soit une lecture dans ce tampon. Lors de la génération d'exécutifs il est donc indispensable de garantir que l'exécution d'une opération consommatrice commence toujours après la fin de l'exécution de ou des opérations productrices. On rappelle que chaque dépendance de données correspond à une précédence d'exécution. Dans le cas d'une architecture monoprocesseur, l'ordre d'exécution entre les opérations est garanti par leur exécution séquentielle, sans préemption (car si il y a préemption l'ordre d'exécution pourrait ne plus être garanti), par l'unique opérateur du graphe d'architecture. C'est donc le respect des dépendances de données lors de la mise en séquence (construction du graphe d'implantation) des opérations et l'unicité du séquenceur dans le graphe d'architecture, qui garantissent la validité de l'exécution de l'algorithme dans le cas d'applications monoprocesseur par conservation de l'ordre partiel $\preceq$ du graphe d'algorithme initial.

6.1.2 Réseaux de Pétri

Introduction

Nous modélisons le comportement macroscopique (séquencement d'opérations plutôt que d'instructions) du séquenceur du processeur à l'aide d'un réseau de Pétri [12, 11, 16]. En effet, ces derniers sont bien adaptés pour modéliser le comportement dynamique des systèmes discrets en tenant compte des ressources qu'ils utilisent. Cela permet de vérifier dans le cas des architectures multiprocesseur, vues plus loin, que l'accès aux tampons mémoire partagés entre les processeurs lorqu'ils doivent communiquer, se fait en exclusion mutuelle. Cette vérification sera faite sur le graphe d'implantation sur lequel sont ajoutées des opérations de synchronisation.

Définitions Ces réseaux reposent sur deux types d'objets : *les places* et *les transitions*. A chaque place est associé un nombre entier de marques. L'état du système est alors associé à un marquage définissant pour toute place le nombre de marques qui lui est affecté. L'ensemble des transitions représente l'ensemble des événements dont l'occurrence provoque la modification de l'état du système, c'est à dire le franchissement d'une transition. Chaque franchissement dépend de la satisfaction de *pr´econditions* qui portent sur le nombre de marques contenues dans les places d'entrée de la transition associée à l'évènement considéré. Le franchissement d'une transition modifie le marquage : toutes les marques qui ont satisfait la transition sont retirées des places d'entrée de la transition et des marques sont ajoutées dans ses places de sortie.

Un réseau de Pétri [12]est défini par un quadruplet $R = (P, T, Pr, Post)$, où :

$P = P_k, k = 1, 2, ..., m$ est un ensemble fini de places,

$T = T_i, i = 1, 2, ..., n$ est un ensemble fini de transition,

$Pr : P \times T \rightarrow I\!N$ est l'application d'incidence avant,

$Post : T \times P \rightarrow IN$ est l'application d'incidence arrière.

Un réseau de Pétri marqué est un couple $N = (R, M)$ où R est un réseau de Pétri et $M : P \rightarrow IN$ un marquage. M peut être représenté par un vecteur appelé vecteur d'état. $M(k)$ est le marquage de la place P_k (nombre de marques contenu dans P_k).

Un réseau de Pétri peut être considéré comme un graphe biparti, orienté et valué dont l'ensemble des sommets est $P \cup T$. La valuation des arcs est donnée par les applications Pr et $Post$:

si $Pr(k,i) > 0$, l'arc (P_k, T_i) existe et est valué par $Pr(k,i)$,

si $Post(k,i) > 0$, l'arc (T_k, T_i) existe et est valué par $Post(k,i)$,

Règle Une transition t est franchissable pour un marquage M si et seulement si $\forall p \in P, M(p) \geq Pr(p,t)$. Si t est franchissable pour M, le nouvel état M' obtenu après le franchissement effectif de t est $M'(p) = M(p) - Pr(p,t) + Post(p,t)$. En terme de graphe, t est franchissable pour M si et seulement si chaque place p entrée de t contient un nombre de marques supérieur à la valuation de l'arc (p,t). Le franchissement de t consiste alors pour chaque place p entrée de t à enlever $Pr(p,t)$ marques et pour chaque place p sortie de t à ajouter $Posr(t,p)$ marques.

Représentation Habituellement, les places sont représentées par des cercles et les transitions par des rectangles (Cf. figures 6.1 et 6.3) et les application Pr et $Post$ par des arcs valués. La valeur 1 sera omise par la suite, et l'absence d'arc indique la valeur 0. Les marques sont représentées par des points à l'intérieur des places, et sont souvent appelés *jetons*, terme que nous utiliserons toujours par la suite.

Utilisation Dans le cadre de cette thèse nous allons utiliser les réseaux de Pétri pour décrire le déroulement de l'exécution de l'algorithme sur l'architecture au niveau de l'allocation des séquenceurs de calculs (opérateurs) et des séquenceurs de communications (communicateurs). Nous construisons le réseau de Pétri d'une implantation à partir des opérations distribuées et ordonnancées sur les ressources séquentielles du graphe d'architecture. Chaque opération du graphe d'implantation correspond à une transition du réseau, et chaque précédence (avec ou sans données) entre opérations correspond à une place. Une transition (exécution d'une opération) est franchie quand tous les jetons de ses places d'entrées sont présents (les prédécesseurs de l'opération ont été exécutés). La présence de jetons dans les places modélise donc à la fois les données produites et consommées par chaque opération et les données de contrôle de l'ordonnancement. Ces dernières permettent de modéliser les synchronisations entre séquenceurs comme nous le verrons dans le paragraphe 6.2.2. Les sommets allocation et identité du graphe d'implantation ne sont pas utilisés ici puisqu'ils ne correspondent pas à des opérations. Dans la section suivante nous étendrons les règles de constructions de réseaux de Pétri aux graphes d'exécution multi-opérateurs.

La figure 6.1 présente un exemple simple de graphe d'implantation dans le cas d'une application basée sur une architecture mono-opérateur, et le réseau de Pétri correspondant. L'ordre partiel du graphe d'algorithme initial à été rendu total grâce à l'ajout de la précédence (sans données) qui est représentée par un arc pointillé. Le séquenceur exécute d'abord l'opération `Cst` , puis il exécute `capte` , `calc` , et `cmd` .

6.1.3 Phases d'une exécution

Les systèmes informatiques possèdent intrinsèquement un certain degré de flexibilité. Il est donc souvent nécessaire de les adapter aux besoins spécifiques de l'application par une phase préalable de configuration. Ainsi, avant d'exécuter la partie itérative de l'algorithme (i.e. l'algorithme spécifié sous la forme de graphe de dépendances infiniment factorisé), il faut exécuter un certain nombre d'opérations d'initialisation ou de configuration (par exemple des périphériques d'entrées sorties) préalable à l'exécution de l'algorithme. De plus, si la durée de vie de l'application n'est pas infinie (cas de la phase de mise au point), elle peut être suivie par des opérations de collecte de données ou des opérations plus spécifiques. Le programme d'un processeur est donc constitué de trois parties qui correspondent à trois phases successives :

- *une phase d'initialisation,*

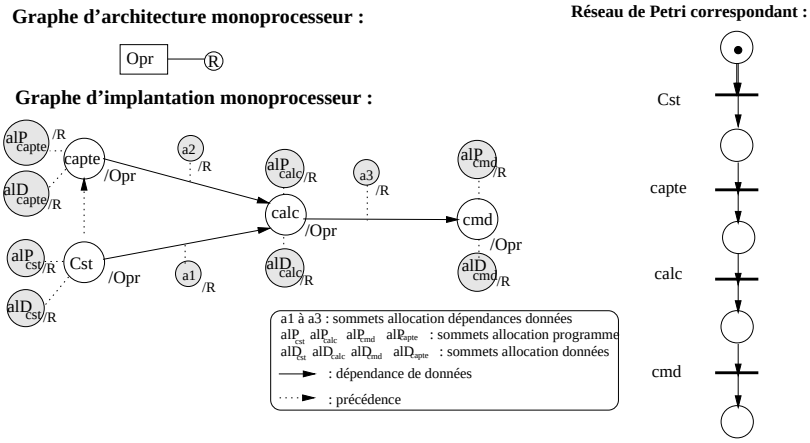


FIG. 6.1: *Réseau de Pétri construit à partir d'un graphe d'exécution monoprocesseur*

- *une phase d'it´eration,*
- *une phase de fi nalisation.*

Les applications que nous traitons étant réactives (interraction avec l'environnement à travers une itération infinie), elle sont toujours basées sur la séquence suivante répétée infiniment :

1. acquisitions des données (lectures des capteurs),
2. calculs,
3. productions des sorties (commandes des actionneurs).

6.1.4 Phase d'itération

Comme la spécification de l'algorithme décrit uniquement la partie infiniment itérative de l'application sous une forme factorisée, il est nécessaire de transformer cette spécification de façon à pouvoir y ajouter les phases d'initialisation et de finalisation composant tout programme d'un processeur.

La première transformation consiste donc à extraire le motif infiniment répété du graphe d'implantation. Ensuite, il faut ajouter de nouveaux sommets au graphe extrait précédemment, afin de le rendre à nouveau répétitif, tout en permettant maintenant de spécifier les parties non répétitives correspondant aux phases d'initialisation et de finalisation. Cette transformation du graphe d'un motif en un graphe d'exécution, consiste à ajouter un sommet spécial, appelé `LOOP` , après la dernière opération *constante* ordonnancée sur l'opérateur (Cf. exemple figure 6.2). En effet, les opérations constante n'ayant pas d'entrée et générant les même valeurs de sortie à chaque exécution, il n'est pas nécessaire de les exécuter à chaque itération de l'algorithme. Ainsi, pour que la ressource processeur ne soit pas gaspillée en les exécutant inutilement à chaque itération, il est préférable de les exécuter une seule fois avant l'exécution de la phase itérative. C'est pourquoi, lors de la construction du graphe d'implantation (Cf.4.2.3.4, P. 117), toutes les opérations *constantes* ont été ordonnancées de façon à être les premières exécutées. Symétriquement, pour marquer la fin de la phase itérative et le début de la phase de finalisation, nous ajoutons un second sommet, appelé `ENDLOOP` , après la dernière opération ordonnancée sur l'opérateur. Ces deux nouveaux sommets appartiennent à l'ensemble des opérations systèmes noté O'_{sys} et sont connectés uniquement par des arcs de précédence $\bar{D}''_P$ (puisqu'ils ne transportent pas de données). Ainsi, après exécution du sommet `ENDLOOP` , l'opérateur

exécute, soit le sommet qui le suit si une certaine condition (par exemple passage par zéro d'un compteur d'itération) est remplie, soit il exécute le sommet LOOP (qui décrémente le compteur d'itération) si elle ne l'est pas. Dans le cas d'une répétition infinie, l'opération exécutée après le sommet ENDLOOP est toujours l'opération LOOP .

Les sommets ordonnancés avant le LOOP font partie de la phase d'initialisation. Les sommets ordonnancés entre le LOOP et le ENDLOOP font partie de la phase d'itération, et les sommets ordonnancés après le ENDLOOP font partie de la phase de finalisation.

Propriété 4 *La factorisation explicite du graphe d'exécution par l'insertion des sommets systèmes* LOOP *et* ENDLOOP *ne modifie pas son ordre partiel puisque les dépendances entre opérations sont toutes conservées.*

6.1.5 Phases d'initialisation et de finalisation

Pour chaque opération d'entrée-sortie (type extio), nous créons et ajoutons un sommet d'initialisation correspondant dans la phase d'initialisation et un sommet de finalisation correspondant dans la phase de finalisation. Soit O'_{init} et O'_{end} les ensembles d'opérations d'initialisation et de finalisation. Ces opérations sont connectées par des arcs de précédence $\bar{D}''_P$ car il ne consomment pas et ne produisent pas de données. Ces sommets ont pour but d'effectuer les initialisations (configuration des ports du processeur en entrée ou en sortie, etc) préalables à l'exécution des opérations d'entrées ou de sorties. Chaque sommet d'initialisation et de finalisation d'entrée-sortie est étiqueté par le nom des opérations d'entrée-sortie effectives auxquelles elles correspondent, suffixé par les mots "_ini" pour les opérations d'initialisation, ou "_end" pour les opérations de finalisation.

Les sommets d'initialisation sont ordonnancés avant les opérations constantes, et si il n'en existe pas, avant le sommet LOOP puisque ce dernier marque le début de la phase itérative. Les sommets de finalisation sont ajoutés après le sommet ENDLOOP puisqu'ils marquent la fin de la phase itérative et le début de la phase de finalisation. Comme au sein d'une même phase d'initialisation ou de finalisation les sommets ne sont pas en dépendance de données, il est nécessaire d'ajouter des arcs de précédence entre chacun de ces sommets.

Propriété 5 *Les sommets d'initialisation (O'_{init}) et de finalisation (O'_{end}), ainsi que les sommets systèmes (O'_{sys}) sont ajoutés avant la première opération du motif du graphe d'implantation factorisé, et après la dernière opération de ce graphe auquel ils sont connectés par des arcs de précédence $\bar{D}''_P$ intra-opérateur. L'ordre total intra-opérateur n'est donc pas modifié, ainsi que l'ordre partiel $\trianglelefteq$ entre toutes les opérations du graphe d'implantation. Cette transformation, modélisée par la relation it´eration-mono-op´erateur $\mathcal{R}_{IterMono}$, conduit à un graphe d'exécution :*

$$(G_{reallocR}, G'_{ar})(.) \xrightarrow{\mathcal{R}_{IterMono}} (G_{exec}, G'_{ar})(.)$$

$$\begin{array}{r} (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \\ \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}}, \\ H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}) \end{array} \xrightarrow{\mathcal{R}_{IterMono}} \begin{array}{l} (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \\ \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}} \\ \cup O'_{init_{CAL}} \cup O'_{end_{CAL}} \cup O'_{sys_{CAL}}, \\ H''_\alpha \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_P \cup \bar{D}''_C \cup D^*_{PC}) \end{array}$$

Les instructions composant les opérations d'initialisation et de finalisation sont spécifiques à chaque processeur, c'est pourquoi ils ne doivent pas apparaître sur le graphe d'algorithme qui deviendrait spécifique à une architecture donnée. Ces sommets n'étant exécutés qu'une seule fois, lors de l'initialisation ou lors de la finalisation, ils ne modifient pas la durée d'exécution d'une itération de l'algorithme, c'est pourquoi nous les ajoutons après l'optimisation sur le graphe d'exécution.

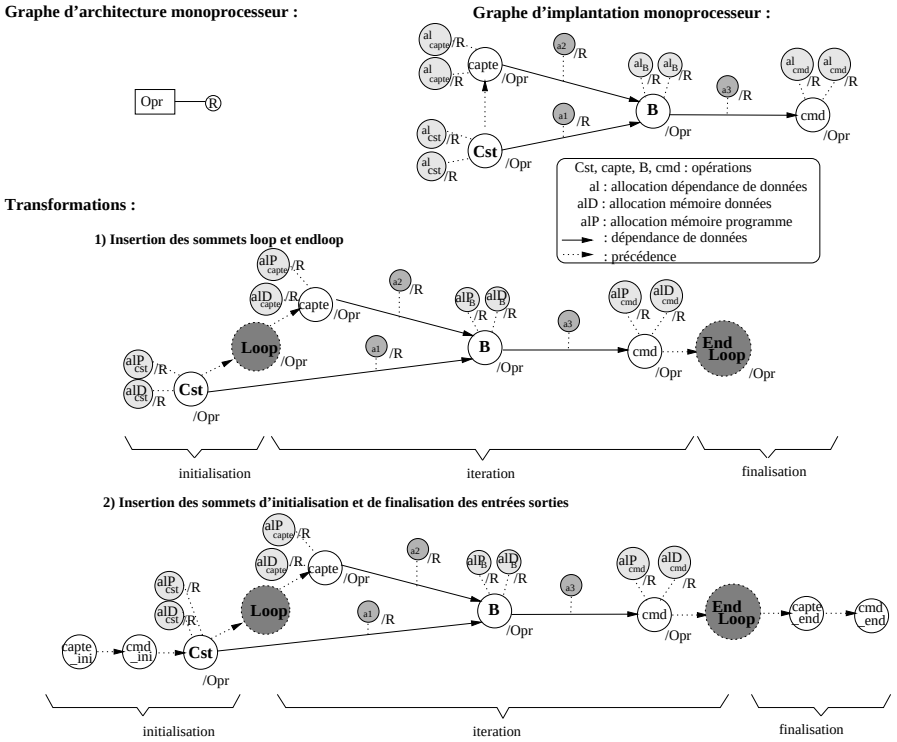


FIG. 6.2: *Exemple de graphe d'exécution d'une architecture monoprocesseur*

A partir du graphe d'exécution qui vient d'être construit, il est possible de construire le réseau de Pétri équivalent pour décrire le comportement dynamique de l'application. Le sommet LOOP corrrepsond alors à une transition qui possède deux prédécesseurs, le premier n'étant exécuté que la première fois (il appartient à la phase d'initialisation), le second étant la transition ENDLOOP. Cette dernière possède deux successeurs : une transition t_v (dont l'unique successeur est une transition de la phase de finalisation) et une transition t_f dont l'unique successeur est la transition LOOP. Les transitions ENDLOOP, t_v,t_f et la place qui les connecte forment un "sous-réseau de test" [11] : la transition LOOP est associée à l'évaluation d'un prédicat (condition d'arrêt de l'application), lorsque cette transition est franchissable le prédicat associé est évalué. En fin d'évaluation, t_v est franchissable si le prédicat est vrai, sinon c'est t_f qui est franchissable. Il est possible d'expliciter le réseau dans lequel la décision a été prise, (Cf. partie droite de la figure 6.3), mais cela surcharge sensiblement la lecture du réseau de Pétri.

Quelque-soit l'application monoprocesseur, le graphe d'exécution est toujours composé de trois phases et de sommets spécifiques à chacune d'elles, le réseau de Pétri correspondant a donc une structure identique. L'utilisation des réseaux de Pétri nous permet de montrer qu'il n'y a jamais famine (manque de ressource), ni accumulation de jetons dans le réseau durant l'exécution de l'algorithme, il ne peut donc jamais y avoir blocage.

Exemple 6.1.1 *La figure 6.3 présente le réseau de Pétri du graphe d'exécution de la figure 6.2.*

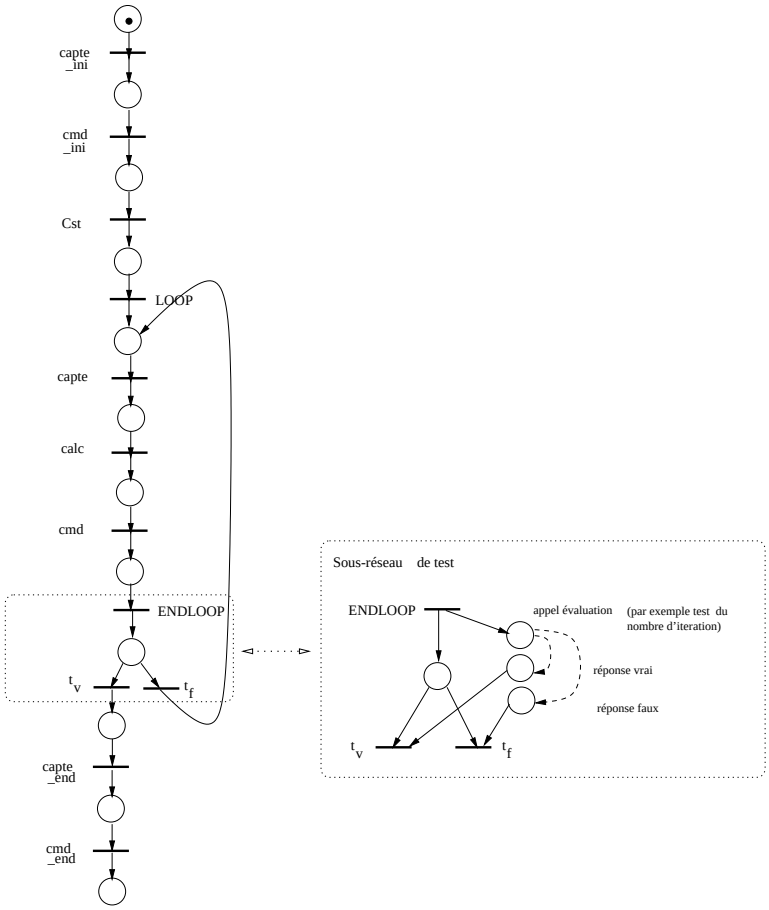


FIG. 6.3: *Réseau de Pétri correspondant au graphe d'exécution explicitement factorisé de l'exemple*

6.2 Multiprocesseur

6.2.1 Séparation des phases

Dans le cas d'architectures multiprocesseur, il existe autant de séquences de calcul que de processeurs (puisque chaque processeur renferme un opérateur) et autant de séquences de communication que de communicateurs dans le graphe d'architecture. Les nécessités d'initialisation et de finalisation exposées dans le précédent paragraphe restent vraies dans le cas de ces architectures. Ainsi, pour chaque processeur, il faut séparer les différentes phases d'exécution par l'insertion de sommets `LOOP` et `ENDLOOP` sur chaque opérateur, mais aussi sur chaque communicateur. En effet la séquence d'opérations de communication associée à chaque opérateur est elle aussi répétée infiniment (ou un nombre limité de fois pour les raisons exposées plus haut). De plus, nous verrons que dans la phase d'initialisation des communicateurs il sera nécessaire d'initialiser l'état des sémaphores utilisés pour synchroniser les communications (Cf. explication de la section suivante), et d'effectuer un certain nombre d'opérations de configurations spécifiques à chaque communicateur. Enfin, dans la phase de finalisation des communicateurs, il sera parfois nécessaire d'ajouter des sommets pour collecter des traces d'exécution.

Dans le cas d'une application multiprocesseur, il est donc nécessaire de commencer par effectuer les mêmes transformations du graphe d'implantation (relation $\mathcal{R}_{IterMono}$) que dans le cas monoprocesseur. Sur

chaque opérateur et communicateur il faut donc commencer par séparer les trois phases par des sommets LOOP et ENDLOOP , puis ajouter les sommets d'initialisation et de finalisation correspondant aux opérations d'entrée-sortie mais aussi des communicateurs eux-même. Il peut en effet être nécessaire d'effectuer certaines initialisations au niveau de chaque communicateur (configuration des liens, etc). Le nom de chaque sommet d'initialisation et de finalisation d'un communicateur est obtenu à partir du nom du communicateur, suffixé par "_ini_" en phase d'initialisation et "_end_"en phase de finalisation. Ces sommets sont ajoutés de façon à être respectivement les premiers et les derniers exécutés dans chaque séquence de communication.

Exemple 6.2.1 *La figure 6.4 présente un exemple de graphe d'exécution obtenu après ces transformations du graphe d'implantation. Les sommets grisés représentent les sommets* loop *et* endloop *ajoutés sur chaque opérateur.*

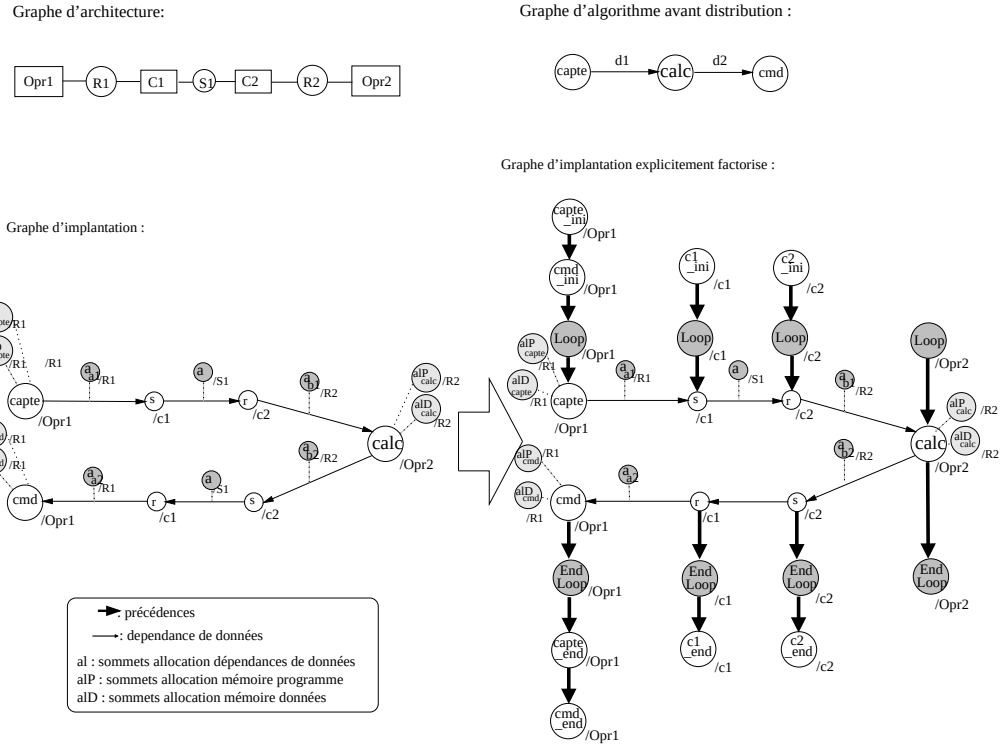


FIG. 6.4: *Graphe d'exécution factorisé explicitement d'une application multiprocesseur*

Propriété 6 *Comme dans le cas monoprocesseur, les sommets d'initialisation (O'_{init}) et de finalisation (O'_{end}), ainsi que les sommets systèmes (O'_{sys}) sont ajoutés sur chaque opérateur avant la première opération du motif du graphe d'implantation factorisé, et après la dernière opération de ce graphe auquel ils sont connectés par des arcs de précédence $\bar{D}''_P$, intra-opérateur. Dans le cas multiprocesseur, ces sommets doivent aussi être ajoutés sur les communicateurs, ils sont connectés aux autres sommets par des arcs de précédences intra-communicateur $\bar{D}'''_C$. Cette transformation est modélisée par la relation it´eration $\mathcal{R}_{iter}$:*

$$(G_{reallocR}, G'_{ar})(.) \xrightarrow{\mathcal{R}_{Iter}} (G_{exec}, G'_{ar})(.)$$

$$(O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}}, H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC}) \xrightarrow{\mathcal{R}_{Iter}} (O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}} \cup O'_{init_{CAL}} \cup O'_{end_{CAL}} \cup O'_{sys_{CAL}}, \cup O'_{init_C} \cup O'_{end_C} \cup O'_{sys_C}, H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_P \cup \bar{D}''_C \cup \bar{D}'''_C \cup D^*_{PC})$$

6.2.2 Synchronisation

6.2.2.1 Principes

Nous avons vu, dans la première section de ce chapitre, que l'ordre d'exécution spécifié par les dépendances de données ou par les précédences étaient réalisées par la mise en séquence des opérations dans le graphe d'implantation. L'unicité du séquenceur de l'architecture et l'absence de préemption garantissent qu'une opération consommatrice de données est toujours exécutée après la fin de l'exécution de ou des opération(s) productrice(s) de ces données.

Dans le cas d'applications multiprocesseur, l'ordre d'exécution entre les opérations d'une même partition est toujours garanti puisque chaque élément de partition (sous graphe associé d'ordre total) n'est associée qu'à un seul séquenceur fonctionnant sans faire de préemption. Par contre, tel quel, rien ne garantit l'ordre partiel d'exécution (dépendances $D_p{}^*$) entre des opérations exécutées par des séquenceurs différents. Nous allons donc maintenant étudier comment garantir et réaliser l'ordre partiel d'exécution spécifié dans le graphe d'implantation. Cette réalisation est différente selon que la mémoire partagée par les séquenceurs est de type RAM ou SAM.

6.2.2.2 Mémoire RAM

Rappelons qu'une RAM (Cf. chapitre "Modèle d'architecture") est une mémoire indexable à accès aléatoire, c'est-à-dire permettant d'y lire les données dans un ordre différent de celui où elles y ont été écrites. Cette différence possible d'ordre entre écriture et lecture permet un décalage temporel quelconque entre l'écriture d'une donnée et sa lecture plus tard : pour cette raison, les communications par RAM sont souvent dites "asynchrones", qualificatif que nous éviterons d'employer ici d'une part pour sa confusion possible avec les qualificatifs "synchrone" ou "asynchrone" relatifs aux protocoles matériels d'échanges des données sur les bus connectant mémoires et processeurs, et d'autre part parce qu'il est question ici uniquement de communications *synchronis´ees.*

Par rapport à un tampon mémoire alloué sur une RAM partagée entre les séquenceurs d'un opérateur et d'un communicateur (ou entre deux séquenceurs de communicateurs), appelons "producteur" l'opération qui y écrit, exécutée par l'un des deux séquenceurs, et "consommateur" l'opération qui y lit, exécutée par l'autre séquenceur. Il ne faut pas qu'un tampon soit lu avant qu'une donnée y ait été écrite (Cf. figure 6.5).

De plus, comme la factorisation est supposée infinie, à chaque itération nous réutilisons le même espace mémoire (tampon) pour communiquer les données entre les opérations, tant que les opérations productrice et consommatrice de données sur ce tampon sont identiques, cela ne pose pas de problème. Dans le cas de tampons partagés entre des opérations exécutées par des séquenceurs différents il faut prendre des mesures pour ne pas écraser une donnée du tampon par une nouvelle donnée tant qu'elle n'a pas été lue par tous les consommateurs qui l'utilisent (il peut en effet y avoir plusieurs consommateurs en cas de diffusion, Cf. figure 6.5).

Il faut donc garantir deux types de synchronisation entre producteur et consommateur(s) :

– le premier type correspond directement à l'implantation des précédences tirées des dépendances de données spécifiées dans le graphe d'implantation. Elles garantissent que l'exécution du producteur se termine avant le début de l'exécution du consommateur : nous appellerons ces précédences *tampon-plein* (puisque le tampon doit être plein pour que ce qui suit soit exécuté). Puisque ces synchronisations assurent un ordre d'exécution entre des opérations exécutées au sein d'une même itération, elles appartiennent à la classes des *synchronisations intra-it´eration*,

– le second type de synchronisation garantit que l'exécution du consommateur se termine toujours avant le début de l'exécution suivante du producteur, lors de l'itération suivante. Nous les appellerons *pr´ec´edence tampon-vide*, elles appartiennent à la classe des *synchronisations inter-it´eration.*

Remarque 35 *Ce schéma de synchronisation est très général, il se transpose directement au niveau matériel dans le cas des circuits dits "asynchrones", où les précédences tampon-plein et tampon-vide sont habituellement dénommées "data set ready" ou "strobe" pour la première, et "data acknowledge" ou "data request" pour la seconde.*

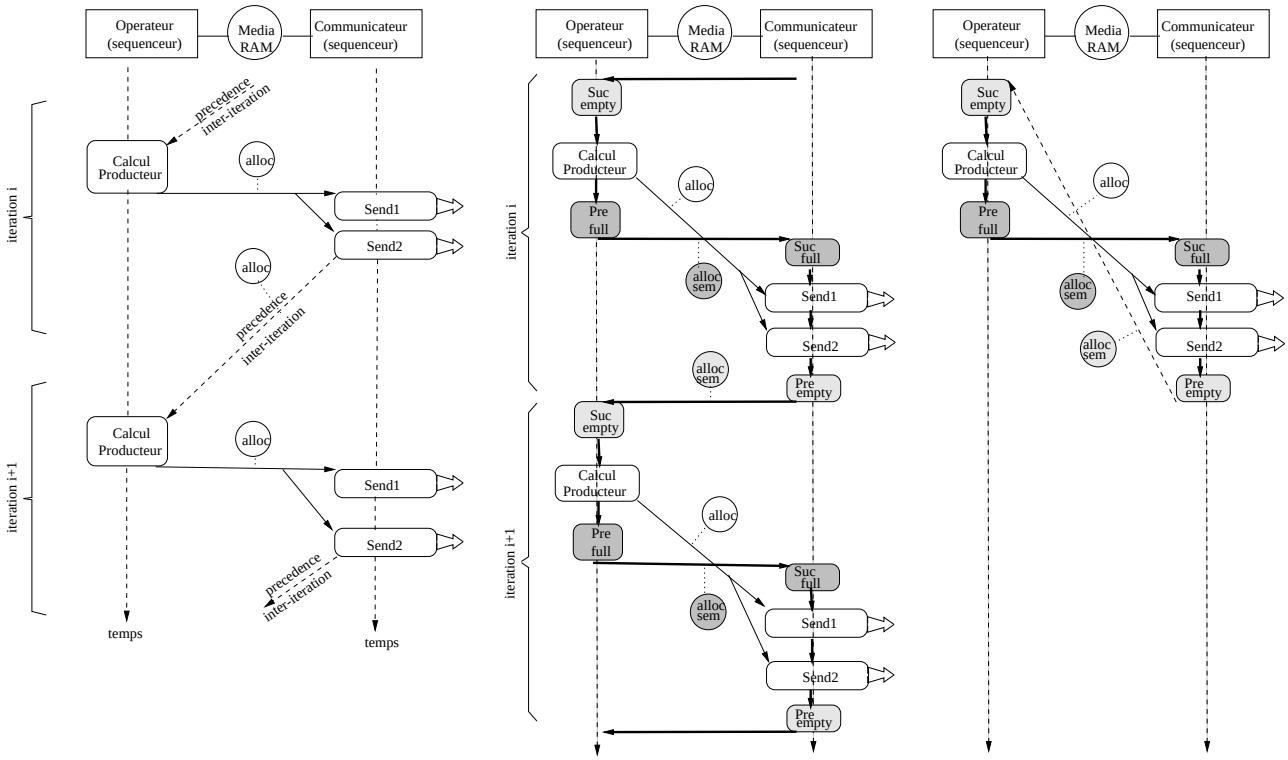
Une précédence est réalisée à l'aide de deux opérations spéciales connectées par une dépendance de données. Elles sont distribuées sur les deux séquenceurs qui exécutent les opérations à synchroniser, la dépendance de données induit un sommet allocation mémoire qui sera distribué sur la RAM partagée afin de faire communiquer les deux opérations. Appelons `Pre` l'opération de synchronisation distribuée sur le séquenceur du producteur dont l'exécution doit *pr´ec´eder* celle du consommateur. Appelons `Suc` l'opération de synchronisation distribuée sur le séquenceur du consommateur dont l'exécution *succède* à celle du producteur. Soit O'_{sync} l'ensemble des opérations de synchronisation.

L'exécution du `Suc` ne doit pas se terminer avant la fin du `Pre` correspondant, par contre `Pre` peut se terminer avant la fin du `Suc` correspondant. Donc `Suc` est *bloquant* (si il est exécuté plus tôt il doit attendre l'exécution de `Pre`) alors que `Pre` est *passant* (si il est exécuté plus tôt, il peut se terminer sans attendre).

Remarque 36 *En termes de sémaphores,* `Pre` *et* `Suc` *correspondent respectivement aux opérations indivisibles* `V` *de libération et* `P` *de prise du sémaphore, aussi appelées respectivement* `Signal` *et* `Wait` *dans les exécutifs traditionnels.*

L'implantation, sur le graphe d'exécution, des deux types de synchronisations (intra-itération et inter-itération) correspond à deux types de transformations (Cf. figure 6.5) :

1. Réalisation des précédences intra-itération (tampons-pleins) :
   - après chaque opération productrice, il faut ajouter autant de sommets `Pre_full` (tampon plein) qu'il y a de consommateurs de ses données distribuées sur d'autres séquenceurs,
   - avant chaque opération consommatrice, il faut ajouter autant de sommets `Suc_full` qu'il y a de producteurs des données consommées distribuées sur d'autres séquenceurs,
   - il faut ajouter autant de sommets `alloc_sem_full` , sur les RAM partagées par les séquenceurs, qu'il y a de couples (`Pre_full` , `Suc_full`).
2. Réalisation des synchronisations inter-itération (tampon-vide) :
   - avant chaque opération productrice, il faut ajouter autant de sommets `Suc_empty` qu'il y a de consommateurs de ses données distribuées sur d'autres séquenceurs,



a) précédences entre producteur et consommateurs

b) insertion des sommets de synchronisation sur deux itérations du graphe d'exécution

c) sommets de synchronisation sur graphe d'exécution (ce graphe n'est pas representé entièrement)

FIG. 6.5: *Réalisation des synchronisations (les schémas ne représentent qu'une partie du graphe d'exécution afin de montrer uniquement les synchronisations entre opérateur et communicateur)*

- après chaque opération consommatrice, il faut ajouter autant de sommets `Pre_full` (tampons-pleins) qu'il y a de producteurs de ses données distribuées sur d'autres séquenceurs,
- il faut ajouter autant de sommets `alloc_sem_empty` , sur les RAM partagées par les séquenceurs, qu'il y a de couples (`Pre_empty` , `Suc_empty`).

Propriété 7 *Soit $\mathcal{R}_{synchro}$ la relation qui à partir d'un graphe d'exécution construit un graphe d'exécution synchronisé :*

$$(G_{exec}, G'_{ar})(.) \xrightarrow{\mathcal{R}_{synchro}} (G_{sync}, G'_{ar})(.)$$

$$\begin{aligned}(O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}} \cup O'_{init_{CAL}} \cup O'_{end_{CAL}} \cup O'_{sys_{CAL}}, \\ H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC})\end{aligned} \xrightarrow{\mathcal{R}_{synchro}} \begin{aligned}(O'_{CAL} \cup O''_{COM} \cup O''_{alloc_{COM}} \cup O'''_{allocP_{MEM}} \cup O'''_{allocD_{MEM}} \cup O''_{ident_{BUS}} \cup O''_{alias_{MEM}} \cup O'_{init_{CAL}} \cup O'_{end_{CAL}} \cup O'_{sys_{CAL}} \cup O'_{sync_{CAL}}, \\ H''_{\alpha} \cup H''_{\alpha_P} \cup H''_{\alpha_D} \cup \bar{D}'_P \cup \bar{D}''_C \cup D^*_{PC})\end{aligned}$$

La réalisation des synchronisations s'effectue par l'ajout de sous-graphes linéaires. L'ordre d'exécution des opérations est nécessairement total dans de tels sous-graphes. Parce que chaque sous-graphe est ajouté

entre les opérations productrice et consommatrice, il ne modifie pas l'ordre d'exécution entre ces opérations. Il y a donc conservation de l'ordre partiel d'exécution entre les opérations de calcul, de communication et d'entrée-sortie.

Remarque 37 *Dans le cas où plusieurs consommateurs utilisant la même donnée sont exécutés dans la même séquence, il suffit d'un seul* `Suc_full` *avant le premier dans la séquence et d'un seul* `Pre_empty` *après le dernier. En effet, puisqu'ils sont exécutés en séquences, seul le premier* `Suc` *suffit à réaliser l'ordonnancement.*

Remarque 38 *Optimisation possible : Une analyse des précédences au niveau global peut également permettre d'éliminer les précédences tampon-vide qui incluent une chaîne causale de précédences tampon-plein.*

Remarque 39 *Ces transformations auraient pu être effectuées lors de l'élaboration du graphe d'implantation mais cela aurait été au détriment de la lisibilité du graphe et surtout d'un surcoût de complexité et de temps d'exécution de l'heuristique d'optimisation. Les prédictions temporelle et mémoire n'en sont pas trop faussées car nous verrons que le codage optimisé de ces sommets induit un surcoût d'exécution négligeable vis à vis des durées d'exécution des opérations de calcul. Nous verrons aussi que l'espace mémoire nécessaire à l'implantation et à l'exécution de ces opérations est lui aussi négligeable. De plus, l'analyse des précédences au niveau global qui conduit à l'élimination d'un grand nombre de précédences tampon-vide ne peut être effectuée qu'après la fin de la construction du graphe d'exécution.*

Une dernière transformation reste à effectuer, il s'agit de l'initialisation des précédences tampon vide. En effet, lors de la première itération de l'algorithme, les précédences `Suc_empty` sont bloquées puisqu'il n'y a pas encore eu exécution des `Pre_empty` correspondant. Pour les débloquer, il faut exécuter, dans la phase d'initialisation de chaque séquenceur, autant de `Pre_empty` qu'il y a de `Suc_empty` dans la séquence. En terme de transformation de graphe d'exécution, cela correspond à l'insertion de sommets `Pre_empty` dans la phase d'initialisation.

Propriété 8 *La modélisation, par un réseau de Pétri (Cf. figure 6.6 et une partie du déroulement de l'exécution figure 6.7), de la synchronisation par les couples de sommets (*`Pre_full, Suc_full`*) et (*`Pre_empty, Suc_empty`*) montrent qu'il n'y a jamais famine ni accumulation de jetons dans les places qui n'ont donc que deux états possibles : vide ou "pleine" (un jeton). L'implantation des synchronisations peut donc s'effectuer à l'aide de sémaphores binaires et garantit une exécution sans deadlock, i.e. que l'accès aux tampons mémoire partagés entre des séquenceurs devant se communiquer des données, se fait en exclusion mutuelle.*

6.2.2.3 Mémoire SAM

Rappelons qu'une SAM est une mémoire à accès séquentiel, c'est-à-dire imposant matériellement que les données y soient lues dans l'ordre où elles y ont été écrites ; cette catégorie comprend les mémoires FIFO (First-In First-Out), et plus généralement toute liaison série ou parallèle, point-à-point ou multipoint, permettant le transfert de données en mode FIFO entre deux ports de communication. Cette identité d'ordre entre écriture et lecture impose d'une part que la i-ième écriture soit terminée avant que la i-ième lecture puisse commencer, et d'autre part, pour une mémoire FIFO de n cellules, que la i-ième lecture soit terminée avant que la $(i + n)$-ième écriture puisse commencer. Pour cette raison, les communications par SAM sont souvent dites "synchrones", qualificatif que nous éviterons d'employer ici pour les mêmes raisons que pour les communications par RAM. Il est préférable de dire que les communications par SAM sont

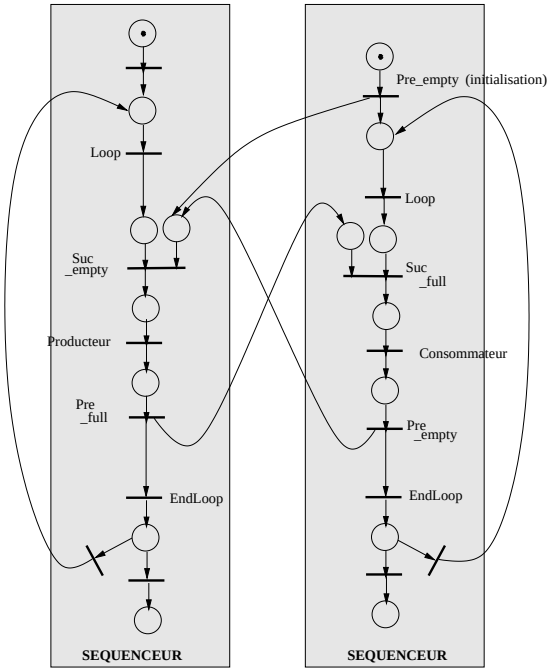


FIG. 6.6: *Sous-réseau de Pétri correspondant à une synchronisation intra-itération (*`Suc _empty` *) et une synchronisation inter-itération (*`Suc _full` *)*

"matériellement synchronisées", ou plus généralement qu'elles sont soumises à un contrôle de flux. Dans le chapitre consacré à la distribution des communications, nous avons considéré deux cas de communication par SAM, selon le rapport de la quantité de données à transmettre et de la capacité de la SAM :

- si la quantité de données à transmettre est inférieure à la capacité de la SAM, une unique paire de sommets `send` et `receive` permet de transférer les données sur la SAM. Les communications par SAM étant matériellement synchronisées il n'est pas nécessaire d'ajouter des sommets de synchronisation, l'opération `receive` commence toujours après la fin de l'opération `send` .
- si la quantité de données à transmettre est supérieure à la capacité de la SAM, nous avons vu que la communication est assurée par l'insertion de plusieurs couples de sommets `send` et `receive` encapsulés. Le sous-graphe correspondant a été factorisé par un couple de sommets `SEND` et `RECEIVE` . Comme le sous-graphe est constitué d'opérations `send -receive` matériellement synchronisées, il est inutile d'ajouter des sommets de synchronisation supplémentaires.

Dans les deux cas, les communications par mémoires SAM ne nécessitent donc pas de modification du graphe car ces communications sont matériellement synchronisées.

Exemple 6.2.2 *La figure 6.8 présente le graphe d'exécution obtenu après transformation du graphe d'exécution de la figure 6.4. Cela correspond à l'insertion des sommets de synchronisation relatif à la communication par RAM (les synchronisations spécifiques aux communications par SAM ne nécessitant pas de transformations). Les dépendances inter-itérations sont représentées en pointillés sur cette figure pour bien souligner que le graphe d'exécution résultant de ces transformations est toujours sans deadlock.*

La construction du réseau de Pétri, équivalent au graphe d'exécution, montre qu'il n'y a jamais famine ni accumulation de jetons dans les places et qu'il est suffisant pour implanter les opérations de synchronisation

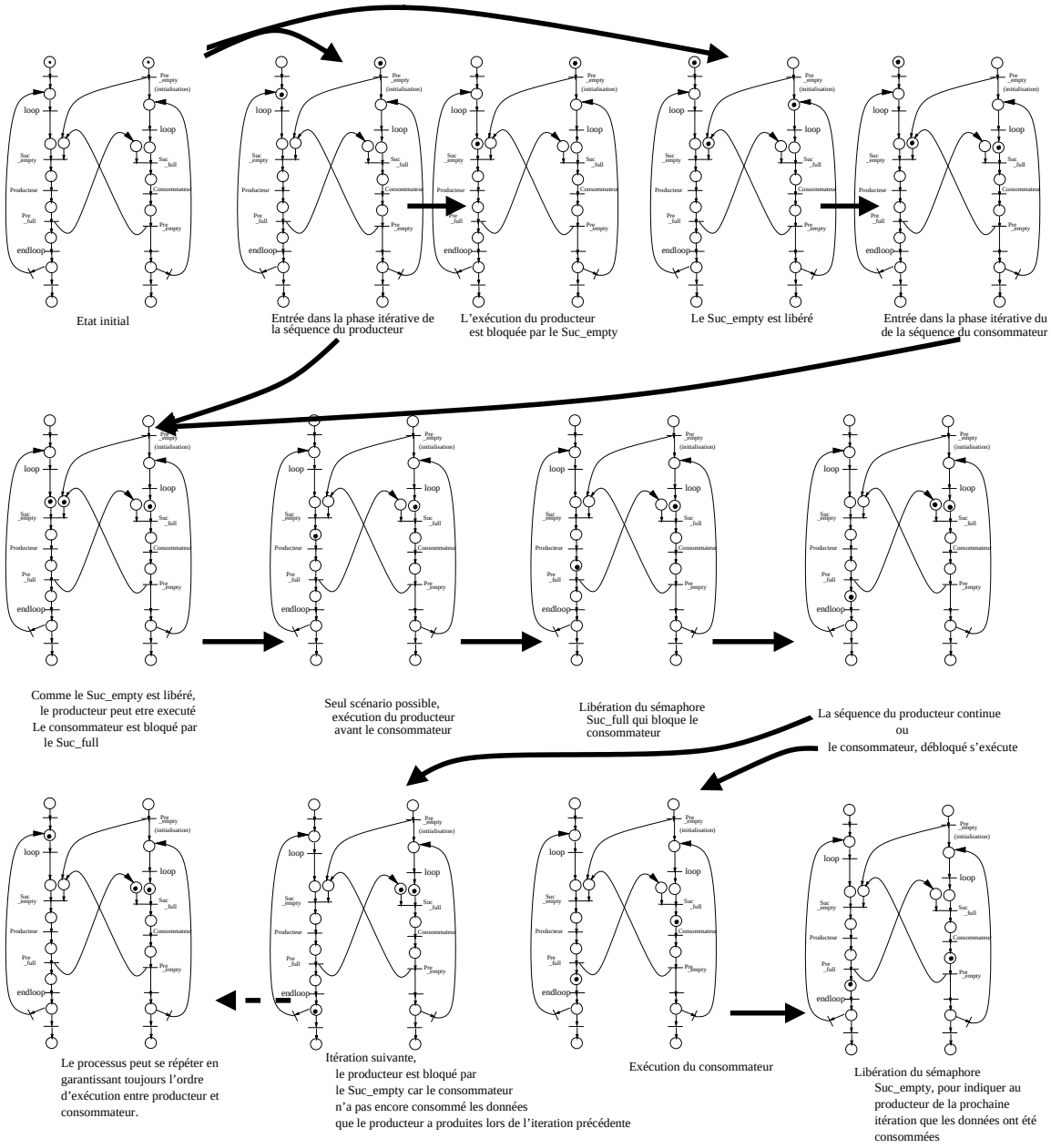


FIG. 6.7: *Déroulement d'une partie de l'exécution*

de manipuler des sémaphores à seulement deux états. La figure 6.9 présente le réseau de Pétri de l'exemple de graphe d'exécution de la figure 6.8.

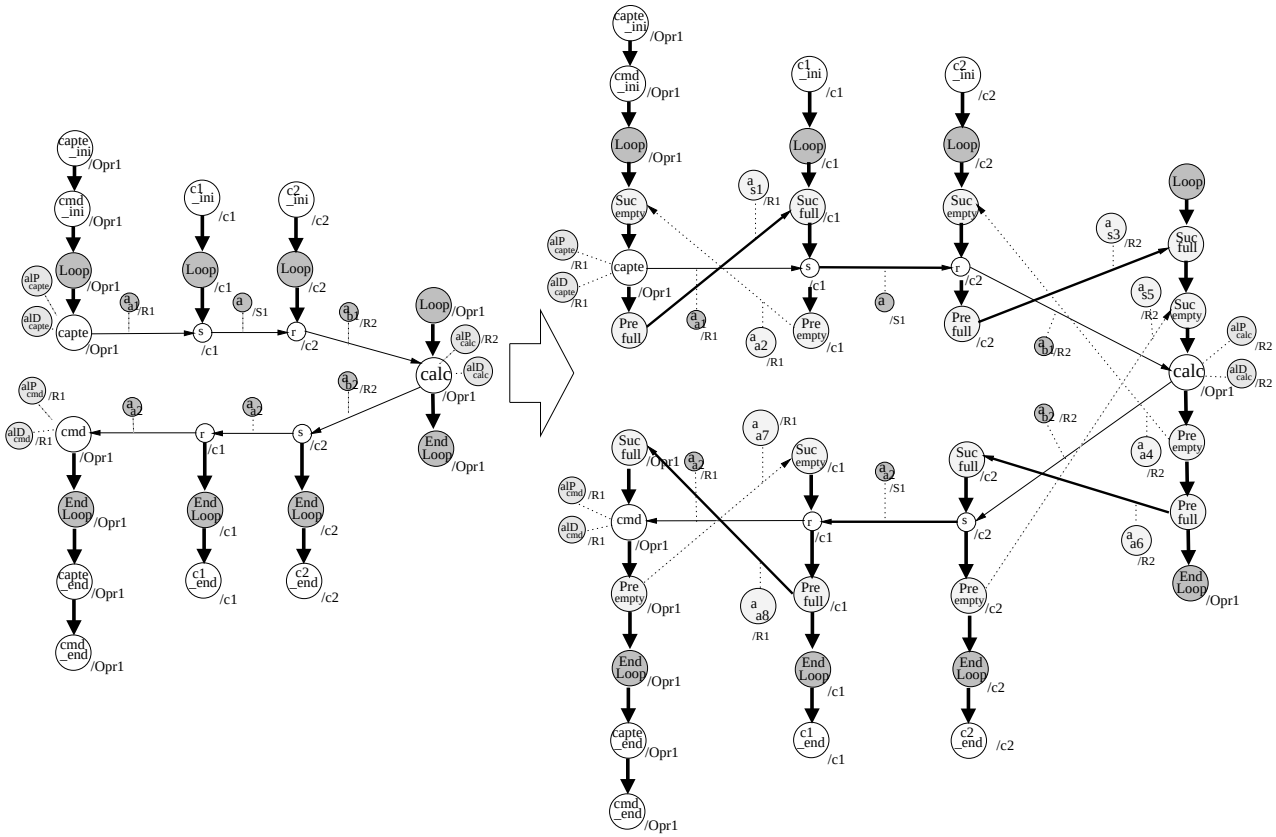


FIG. 6.8: *Graphe d'exécution final*

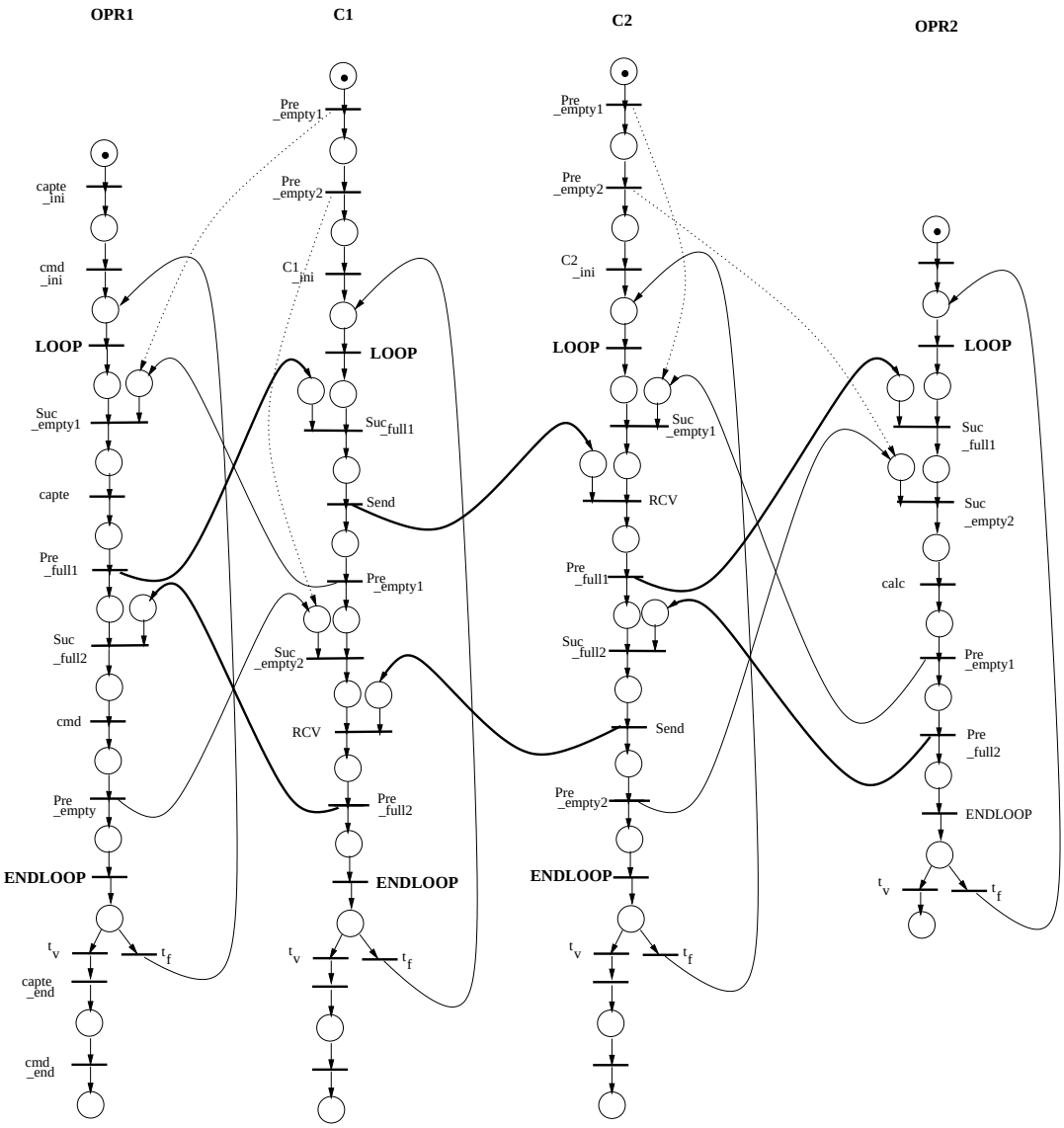


FIG. 6.9: *Exemple de réseau de Pétri d'un graphe d'exécution*

Chapitre 7

Génération de macro-exécutif générique

Sommaire

Dans le précédent chapitre nous avons effectué toutes les transformations du graphe d'implantation en graphe d'exécution synchronisé afin de préparer la génération de code. Nous avons montré (propriétés 1,2,3,4,5, 7) que toutes ces transformations conservaient l'ordre partiel entre les opérations de calcul, d'entrée-sortie et de communication du graphe d'implantation initial. De plus, il est important de souligner que même si la caractérisation des durées d'exécution n'est pas conforme à la réalité et induit donc des prédictions temporelles erronées, seules les optimisations peuvent être faussées dans ce cas. Le graphe d'exécution obtenu reste valide puisqu'il est toujours construit par renforcement de l'ordre partiel spécifié dans le graphe d'algorithme.

Dans ce chapitre nous exposons et justifions la structure du macro-exécutif intermédiaire ainsi que ses règles de construction. Rappelons que la construction d'un macro-exécutif intermédiaire, plutôt que la génération directe d'un exécutif dans un langage donné et fixe, permet de s'affranchir du choix de ce langage, permettant ainsi une plus grande portabilité de l'exécutif ce qui est primordial dans le cas de calculateurs hétérogènes. De plus, cela permet au concepteur de travailler au niveau de spécifications qui convient le mieux à ses objectifs (codage en assembleur plutôt qu'en langage C pour l'optimisation par exemple).

Avant de donner la structure et la syntaxe du macro-exécutif puis ses règles de construction à partir du graphe d'exécution, nous allons commencer par présenter brièvement les principes d'un macro-processeur.

7.1 Macro-processeur

7.1.1 Définition

Un macro-processeur est un programme qui consomme en entrée une chaîne de caractères (texte source), la traite séquentiellement en substituant chaque sous-chaîne qu'il reconnaît ("appel de macro") par une chaîne correspondante de substitution ("définition de macro") qu'il traite à nouveau jusqu'à ce qu'il n'y ait plus de substitution possible, et qui produit en sortie la chaîne de caractères traitée.

Dans un macro-processeur il y a un dictionnaire de "macros" qui associe chaque nom de macro à reconnaître avec une définition. Un appel de macro peut comprendre, juste après le nom de la macro, une liste de sous-chaînes arguments qui est substituée aux paramètres formels trouvés dans la définition de la macro.

Un certain nombre de macros sont prédéfinies au démarrage du macro-processeur dont au moins une qui permet de définir de nouvelles macros.

Nous avons choisi le macro-processeur GNU m4, libre, il a l'intérêt d'être simple mais suffisamment puissant, et d'exister sur toutes les plates-formes ; on le trouve en standard sur les systèmes d'exploitation Unix. Afin de comprendre les exemples qui illustrent ce chapitre, voici ses principales règles de substitution :

– toute chaîne encadrée par un caractère backquote (‘) à gauche et un caractère quote (’) à droite (et pouvant contenir des backquotes et des quotes balancés) est substituée directement, sans nouvelle tentative de substitution, par la même chaîne sans le premier caractère backquote ni le dernier caractère quote.
Par exemple, `‘hello‘’world’` est substituée par `hello‘’world` ,

– les noms de macros ne peuvent être constitués que par des caractères alphabétiques, numériques et du caractère "_" (underscore) et ne doivent pas commencer par un caractère numérique (expression régulière `[_A-Za-z][ _0-9A-Za-z]*`).
Par exemple `foo X1 _1z` sont trois noms de macros possibles,

– les sous-chaînes arguments d'un appel de macro sont séparées par des virgules et leur liste est encadrée entre parenthèses (la parenthèse ouvrante doit être le premier caractère qui suit le nom de la macro).
Par exemple, `foo(un,(2))` appelle la macro `foo` avec deux arguments `un` et `(2)` ,

– la macro `define(name,sub s)` est substituée par une chaîne vide, mais a pour effet de bord de définir une nouvelle macro de nom `name` et de définition `subs` . Pendant une substitution, `$n` sera substitué par le n-ième argument de la macro en cours de susbtitution (et `$0` par le nom de cette macro). Par exemple, `define(‘add’,‘ $0q $1+$2’)add(un, (2))` est substitué par `addq un+(2)` ,

– la macro `dnl` est substituée, ainsi que les caractères qui la suivent jusqu'au premier caractère de fin de ligne suivante (inclus), par une chaîne vide (utile pour commenter et formater les sources).

Les exemples de définition et de substitution des macros, qui illustrent les spécifications des macros, sont imprimés en caractères de style `télétype` .

7.1.2 Règles de dénomination des macros

Pour éviter des conflits de noms entre macros générées, on a suivi les règles de dénomination suivantes :

1. les noms fournis par le concepteur de l'application (pour identifier les sommets du graphe de l'algorithme et de l'architecture, leurs ports, et la macro à générer pour chaque opération) sont constitués d'une chaîne de caractères alphanumériques avec initiale alphabétique (comme dans la plupart des

autres langages, expression régulière [A-Za-z][0-9A-Za -z]*), mais sans caractère “underscore” qui est réservé pour constituer des noms sans conflit avec les premiers,

2. le nom identifiant le tampon mémoire d’une connexion est constitué en concaténant par un “underscore” le nom identifiant le sommet et le nom identifiant le port de sortie à l’origine de la connexion,

3. les macros d’initialisation, d’itération et de finalisation d’une opération d’entrée (sans port d’entrée) ou de sortie (sans port de sortie) sont identifiées en concaténant au nom fourni par le concepteur un “underscore”, un suffixe les différenciant, et un second “underscore”,

4. les macros du noyau générique d’exécutif sont identifiées par un nom suffixé par un “underscore”,

5. les commentaires sont encadrés par des accolades et le mot clé comment , comme dans l’exemple suivant :

```
comment{Ceci    est  un  commentaire}com    me nt
```

7.2 Structure du macro-exécutif

L’exécutif intermédiaire généré pour l’exécutif distribué d’une application est constitué :

– d’un fichier source pour chaque processeur, macro-codant l’exécutif dédié à ce processeur (séquence de calcul, séquences de communications, allocation mémoire etc), qui sera traduit en source compilable pour ce processeur. En effet, dans le chapitre précédent, un processeur a été défini par l’ensemble opérateur (unique) et communicateurs connectés à une même RAM programme,

– et d’un fichier codant la topologie de l’architecture, qui sera traduit en makefile pour automatiser les opérations de compilation.

Comme nous le verrons dans la section suivante, la structure du macro-exécutif intermédiaire généré pour chaque processeur est directement issue de la distribution et de l’ordonnancement des calculs de l’algorithme et des communications inter-processeurs qui en découlent. Chaque fichier d’exécutif commence par les macros :

```
include(syndex.m4x)
processor_(TypeProcesseur,        NomProcesseur,NomApplication)
```

et se termine par :

```
endprocessor_
```

La première macro permet de charger automatiquement un fichier qui définit l’inclusion des bibliothèques (noyaux d’exécutif) contenant les définitions des macros dans le langage du compilateur du processeur cible, nous donnerons la structure de ces noyaux dans le prochain chapitre. Le nom du processeur peut être assimilé au nom de l’opérateur du processeur puisque ce dernier est unique pour chaque processeur. La notion de *type de processeur* permet de factoriser certaines bibliothèques d’exécutif comme nous le verrons en détail dans le prochain chapitre. Le macro-exécutif intermédiaire entre cette paire de macros comprend dans l’ordre :

1. une liste de macros d’allocation mémoire : déclarations de tampons de données, de sémaphores, et optionnellement de tampon de chronométrage,

2. une liste de macros pour chaque séquence de communications (correspondant à chaque communicateur du processeur),

3. une liste de macros pour la séquence de calculs (correspondant à l’unique opérateur du processeur).

7.2.1 Allocation mémoire

Dans ce paragraphe nous définissons les macros qui permettent d'allouer de l'espace mémoire dans une RAM. Avant de les utiliser, il faut toujours indiquer dans quelle mémoire va être alloué cet espace par l'intermédiaire d'une macro RAMarea_(nom_RAM). La fin des allocations dans cette mémoire est marquée par la macro endRAMarea_. Toutes les macros qui vont suivre doivent être générées entre cette paire de macros.

7.2.1.1 Sémaphore

Nous avons vu que chaque couple d'opérations de synchronisation (Pre et Suc) manipule des sémaphores binaires qui sont implantés par des tampons mémoire (sommets allocation). Ces tampons sont déclarés par la macro semaphores_ qui reçoit en argument la liste de noms de sémaphores utilisés dans le graphe d'exécution.

Exemple 7.2.1 *Une implantation possible des sémaphores en langage C consiste à utiliser un tableau d'entiers de dimension égale au nombre de sémaphores, et à utiliser les noms des sémaphores comme indices dans ce tableau en générant automatiquement des "#define NomSemaphore IndiceDansTableau". Par exemple, la définition suivante :*

```
define('semaphores_',      'number_($*)
volatile   int  sem_[$#]={0};    /* les sem. sont initialement    bloqu és*/  ')}
```

réserve un tableau d'entier, tout en générant les "#define" voulus grâce à la définition récursive suivante :

```
define('number_','ifelse($1,,,'
'#define'   $1  decr($#)   number_(shift($*))')')
```

Ainsi, la macro semaphores_(un_empty, un_full) *sera substituée par :*

```
semaphores_(un_empty,      un_full)}
-> #define   un_empty   1
   #define   un_full   0
   volatile   int  sem_[2];
```

Dans le cas où ces sémaphores sont utilisés pour synchroniser des communicateurs communiquant par l'intermédiaire d'une RAM partagée, les sémaphores doivent être alloués dans cette même RAM partagée (i.e. encadrés par les macros RAMarea_ et endRAMarea présentés dans le paragraphe précédent).

7.2.1.2 Tampon mémoire

Chaque tampon mémoire, utilisé pour stocker une dépendance de données (c'est-à-dire un résultat intermédiaire entre deux opérations d'entrée-sortie, de calcul ou de transfert), est déclaré par une macro d'allocation mémoire générique alloc_(type,nom,[taille]) dont les arguments sont :

1. le type de données qui seront stockées dans le tampon,
2. le nom du tampon,
3. la taille du tampon est optionnelle, si elle n'est pas spécifiée sa taille vaut un, (le tampon contient un scalaire).

Rappelons que l'allocation est statique, c'est-à-dire effectuée lors de la compilation, l'adresse et la taille de la zone mémoire allouée restent invariables au cours de l'exécution. Le même nom de tampon est passé en argument :

– à la macro d'allocation,

– à la macro qui code l'opération à l'origine de la dépendance de donnée, qui à l'exécution produira un résultat intermédiaire et l'écrira dans le tampon,

– à chacune des macros qui codent les opérations aux extrémités de la dépendance de données (il peut y en avoir plusieurs en cas de diffusion). A l'exécution, chacune lira le résultat intermédiaire dans le tampon.

Exemple 7.2.2 *Dans le cas de génération de code en C, la plus simple des définitions de la macro* `alloc_` *pourrait être :*

```
define('alloc_',      $1 $2[$3];)}.
```

Ainsi la macro `alloc_(int, i, 10)` *sera substituée par :*

```
alloc\_(int,    i, 10)
->int  i[10];
```

Pour générer du code assembleur on utilisera les directives d'assemblage qui allouent une zone mémoire non initialisée.

Plus précisément, pour permettre des vérifications de cohérences des type des données manipulées par les autres macros mais aussi factoriser les descriptions, la macro `alloc_` est plus complexe et repose sur les macros `typedef_` et `basicAlloc_`:

- tout d'abord, `alloc_`, en plus d'allouer la mémoire, vérifie que le nom de type de ce tampon a bien été défini (grâce à la commande m4 `ifdef` qui permet de générer un message d'erreur à l'aide de la macro `error_` donnée en annexe). Chaque type de données (et leur taille en nombre d'adresses) supporté par le compilateur d'un processeur doit être déclaré dans le noyau spécifique à ce processeur par la macro générique `typedef_(nomType, taille_type)` qui elle fait partie du noyau générique (syndex.m4x, Cf. chapitre suivant) :

  ```
  define('typedef_',      'define('$1_size_',$2)')
  ```

  Remarque 40 *La version suivante de* `typedef_` *vérifie que chaque déclaration d'un nouveau type n'en écrase pas un autre et que la macro reçoit exactement deux arguments (à l'aide de la commande* `ifelse` *de m4) :*

  ```
  define('typedef_',      'ifelse($#,2,'ifdef('$1_size_',
  'error_('WARNING:      type  $1  redefini')')''define('$1_size_',$2)',
  'error_('arguments:      typeName,    typeSizeInAddressUnits')')')
  ```

  Ainsi, dans le noyau d'exécutif spécifique au compilateur C d'un processeur, la définition des types du C et de leur taille s'effectue par :

  ```
  typedef_('bool',      1)
  typedef_('char',      1)
  typedef_('int',       4)
  typedef_('float',     4)
  typedef_('double',8)
  ```

- ensuite, la macro alloc_ vérifie que le type du tampon qu'elle a reçu en argument a bien été déclaré dans l'exécutif spécifique du processeur. Si c'est le cas, alloc_ construit deux macros : NomTampon_type_ et NomTampon_size_ qui seront substitués respectivement par le type et la taille du tampon donné en argument de alloc_. Cela permet à toute macro qui utilise un nom de tampon de connaître la taille et le type des données associées à ce nom de tampon. Bien sûr, avant de les déclarer, alloc_ vérifie que ces macros n'existent pas auquel cas alloc_ est substituée par un message d'erreur :

```
define(‘alloc_’,‘dnl
ifdef(‘$2_type_’,       ‘error_(‘$2    redeclar  é’)’)define(‘$2_type_’,       $1)dnl
ifdef(‘$1_size_’,,      ‘error_(‘Type     non  declar  é:  $1  de  $2’)’)dnl
define(‘$2_size_’,      $3)dnl
basicAlloc_($2)’)

alloc_(int,i,10)
->basicAlloc_(i)
```

- la macro alloc_ n'effectue pas directement l'allocation en C, elle appelle une macro basicAlloc_ qui utilise le fait que chaque tampon a été associé à un type et à une taille, ce qui permet de factoriser les définitions des allocations dans chaque exécutif spécifique. Ainsi, la définition de la macro d'allocation en C est relativement simplifiée :

```
define(‘basicAlloc_’,
$1_type_   $1[$1_size_];)

basicAlloc_(i)
->int  i[10];
```

Les macros typedef_ et alloc_ sont indépendantes du type de processeur et font donc partie du noyau générique d'exécutif (Cf. structure de l'exécutif dans le prochain chapitre). Seule la définition de la macro basicAlloc_ est spécifique au compilateur du processeur cible. Dans cet exemple, le portage de l'exécutif consiste simplement à déclarer les types supportés par ce nouvel exécutif en utilisant typedef_ et à redéfinir la macro basicAlloc_.

Reposant sur le travail effectué par alloc_, la macro type_(NomTampon) permet de connaître le type de chaque tampon car elle est substituée par ce type (ou par le mot “undefined” en cas d'erreur, pour indiquer que le type du tampon n'a pas été défini). Elle sera souvent utilisée par la suite pour vérifier la cohérence des tampons donnés en argument :

```
define(‘type_’,     ‘ifdef($1‘_type_’,$1_type_,‘undefin          ed’)’)  dnl

type_(i)    (si  i  à  été  déclar  é  comme  dans  l’exemple    pr éc édent)
->int
```

Nous utiliserons aussi la macro defined_ pour savoir si un tampon a été déclaré ou non puisque si il a été déclaré par alloc_ il possède un type :

```
define(‘defined_’,      ‘ifdef(‘$1_type_’,,‘error_(‘Undeclared          data  buffer:   $1’)’)’)
```

7.2.1.3 Réallocation de la mémoire

La macro alias_(tampon_réel, tampon_alias) permet de ré-allouer un tampon mémoire déjà déclaré (tampon_réel) afin de le réutiliser pour implanter une autre dépendance de données (tampon_alias) quand il n'y a pas d'intersection dans leur durée de vie (Cf. optimisation mémoire 4).

Exemple 7.2.3 *Lors de la génération de code C, la macro* alias_ *génère simplement une substitution du nom de la variable “tampon_alias” par le nom de la variable qui a été réellement allouée (par une macro*

alloc _. *La définition de cette macro pourrait être (si l'on ne cherche pas à faire de vérification de type) :* define(‘alias _’,‘define(‘$2’ , $1)’) . *Ainsi, après exécution de la macro* alias _(A,B) *toutes les occurrences de* B *sont remplacées par* A.

7.2.2 Séquence de calcul

Comme nous l'avons déjà vu, la séquence d'un opérateur (comme celle d'un communicateur) est composée de trois parties : initialisation, itération, et finalisation. La partie itérative est codée par une liste de macros (une macro pour chaque opération d'entrée-sortie, de calcul, ou de contrôle structuré de répétition ou d'exécution conditionnelle).

7.2.2.1 Structure

1. la phase d'initialisation commence par la macro main _ marquant le point d'entrée après le “boot” du processeur, suivie de macros d'initialisation des constantes et des retards, de macros d'initialisation des interfaces avec l'environnement, d'une macro spawn _thread _ de lancement pour chaque séquence de communication exécutée sur le même processeur (Cf. 7.2.3),

2. la phase itérative comprend des macros d'opérations de calcul et d'interface avec l'environnement exécutées sur le processeur, dont certaines sont précédées et/ou suivies de macros de synchronisation, le tout encadré par deux macros de contrôle itératif loop _ et endloop _,

3. la phase de finalisation comprend des macros de désactivation des interfaces avec l'environnement et se termine par la macro endmain _ marquant la fin de la séquence.

Les macros de calcul et d'entrée-sortie sont spécifiques à l'application et donc spécifiées par le concepteur de l'application, alors que toutes les autres macros forment un *noyau g´en´erique d'ex´ecutif*, jeu de macros indépendant de l'algorithme et de l'architecture de l'application, dont le codage est indépendant de l'algorithme mais dépend du type de processeur cible.

Exemple 7.2.4 *Voici un exemple de définition en langage C des macros* main _ *et* endmain _*:*

```
define(‘main_’, ‘dnl
int main(int argc, char* argv[]) { ’)
define(‘endmain_’, ‘
return 0;
}’)
```

Les macros loop _ *et* endloop _ *peuvent être substituées par une boucle “for” infinie par les définitions suivantes :*

```
define(‘loop_’, ‘for(;;) {’)
define(‘endloop_’, ‘}’)
```

Il est parfois nécessaire de limiter le nombre d'itération de l'algorithme lors de la phase de développement d'une application. Pour cela, la macro loop _ *est implantée différemment, ainsi elle teste l'existence d'une définition de “*NBITERATIONS _*”. Si cette définition existe (car elle a été définie par l'utilisateur, dans le fichier des macros spécifiques à l'application par exemple),* loop _ *est substituée par une boucle finie de* NBITERATIONS _ *itérations :*

```
define(‘loop_’, ‘ifdef(‘NBITERATIONS’, ‘dnl nombre d it ération fini
{int i; for(i=0; i<NBITERATIONS; i++){’, ‘dnl boucle infini
```

```
for(;;){')    ')

define('endloop_',      'ifdef('NBITERATIONS',       '}}',   '}')')
```

Ainsi si NBITERATIONS est défi ni, nous obtenons le code C suivant :

```
define('NBITERATIONS',       512)
loop_
endloop_
->{int   i;  for(i=0;   i<512;   i++){

  }}
```

7.2.2.2 Opération de calcul

Pour chaque sommet opération de calcul du graphe de l'algorithme correspond une définition qui réalise la fonction modélisée par le sommet. Lors de la génération d'exécutif, chaque opération peut être substituée soit par du code inséré en ligne, soit par du code d'appel à une fonction compilée séparément. Dans le second cas, le nom de chaque argument est celui de chaque sommet allocation associé aux dépendances de données connectées à l'opération (chaque argument a donc été préalablement alloué dans une RAM, Cf § 7.2.1.2). L'ordre de ces arguments est fonction des quadruplets associés aux arcs et de la table d'indirection de l'opération présentée dans le chapitre consacré au graphe d'algorithme, nous verrons comment générer correctement cet ordre dans le paragraphe 7.3 consacré à la Génération de macro-exécutif.

Exemple 7.2.5 *Voici trois exemples de définitions de macros arithmétiques et logiques, générant en ligne des expressions C, ainsi qu'un exemple de code généré. Dans l'exécutif C, toutes les données sont considérées comme des tableaux, y compris les scalaires qui sont des tableaux à un seul élément, c'est pourquoi les tampons sont indexés par* `[0]` *dans ces exemples :*

```
define('less','$3[0]=$1[0]<$2[0];')
define('add',     '$3[0]=$1[0]+$2[0];')
define('addVec',      '{  int  i=$1;   while(i--)    $4[i]=$2[i]+$3[i];      }')

add(arg1,    arg2,   sum)
->  sum[0]=arg1[0]+arg2[0];

addVec(10,     v1,   v2,   vsum)
->  {  int   i=10;   while(i--)    vsum[i]=v1[i]+v2[i];       }
```

Lors de la génération de code d'une opération, il est possible, et même recommandé, de vérifier que le type de chaque tampon reçu correspond bien aux types des arguments attendus. A cet effet, nous avons développé la macro générique `mustBe(NomTampon, nomType)`. Si le type du nom de tampon `nomTampon` ne correspond pas au type indiqué par `nomType`, cette macro génère un message d'erreur.

```
define('mustBe','ifelse(index('$2',          $1),-1  ,
'error_('Le    type   $2  est  attendu,    et  non   le  type   $1')')')

mustBe(int,     float)
->error_(Le     type   float   est   attendu,    et   non   le   type   int)
```

Exemple 7.2.6 *La définition de l'opération* `less` *de l'exemple précédent peut ainsi être améliorée de la façon suivante, afin de vérifier par exemple que tous les tampons qu'elle reçoit en arguments sont de type* `int`. *Pour cela on utilise la macro* `type_` *défini dans le paragraphe consacré à l'allocation mémoire :*

```
define('less',
```

```
‘mustBe(type_($1),      ‘int’)’dnl    compare   le  type  du  1er  argument   au  type  int
‘mustBe(type_($2),      ‘int’)’dnl    compare   le  type  du  2nd  argument   au  type  int
‘mustBe(type_($3),      ‘int’)’dnl    compare   le  type  du  2eme  argument   au  type  int
‘$3[0]=$1[0]<$2[0];’)dnl       substitution    de  less  par  son  code  en  ligne
```

Il est aussi possible de vérifier que le nombre d’arguments attendu correspond bien au nombre d’arguments reçu en effectuant un simple test avec la macro `ifelse` de m4 associée à la macro `error_` de notre exécutif générique pour afficher un message d’erreur si le nombre d’arguments ne convient pas.

Exemple 7.2.7 `define(‘less’,‘ifelse($#,3,,‘error_(‘ La macro attend exactement 3 arguments’)’)dnl test nombre d’arguments`
`‘$3[0]=$1[0]<$2[0];’)dnl substitution de less par son code en ligne`

Dans le cas d’appels de fonctions compilées séparément, nous avons développé des macros qui permettent de simplifier l’écriture de chaque macro correspondant à chaque opération. Dans le cas du C, cela correspond à la macro `Ccall_`, basée sur `Cargs_`. La macro `Ccall_(ret,label,arg1, arg2...)` génère un appel à la fonction dont le nom est `label` (deuxième argument). Le premier argument correspond au nom du tampon qui reçoit la valeur retournée par la fonction ou `void` si elle ne retourne rien. Chacun des autres arguments correspond à un nom de tampon et au type de ce tampon, séparés par le caractère espace si ce tampon est un scalaire, ou par le caractère `*` si le passage doit se faire par adresse. Si l’argument est un littéral entier, son type est `const` par convention, le nom étant alors la valeur entière de ce littéral. La macro `Ccall_` utilise la macro `Cargs_` pour analyser le type de chacun de ses arguments.

Exemple 7.2.8 *Voici un exemple d’utilisation de la macro* `Ccall_` *pour générer un appel à la fonction C nommée “seuillage” compilée séparement. Elle reçoit trois arguments : le premier est un entier, les deux autres étant des tableaux, il faut passer leur adresse.*

```
define(‘Seuil’,     ‘ifelse($#,3,,‘error_(‘Expected         3 arguments’)’)dnl
 Ccall_(void,‘seuillage’,       int  $1,  float*$2,    float*$3)dnl    ’)
```

Si les tampons ont correctement été définis par :

```
alloc_(int,    seuil_o,   0)
alloc_(float,gen_o,      512)
alloc_(float,affiche_i,       512)
```

Chaque appel de “Seuil” est substitué par un appel à la fonction C “seuillage” compilée séparément :

```
Seuil(seuil_o,gen_o,affiche_i)
->seuillage(seuil_o[0],       gen_o,   affiche_i);
```

7.2.2.3 Opération d’entrée-sortie

Pour chaque opération d’entrée-sortie du graphe d’algorithme, des sommets d’initalisation et de finalisation ont été ajoutés dans le graphe d’exécution, lors des phases d’initialisation et de finalisation. Le nom de ces sommets est préfixé par le nom de l’opération d’entrée-sortie suffixé de `_ini_` et sans argument pour l’opération d’initialisation et suffixé de `_end_` et sans argument pour l’opération de finalisation. Comme les opérations de calcul, ces opérations seront substituées par du code inséré en ligne ou par du code d’appel à une fonction compilée séparement.

7.2.2.4 Macro de conditionnement

Les opérations conditionnées sont précédées d'une macro : `if_(nom tampon)` et suivies d'une macro `endif_` ou `else_` si le même tampon est utilisé pour conditionner l'exécution d'une autre opération exécutée exclusivement. Dans ce cas la macro `else_` est suivie de cette opération, puis la macro `endif_` est insérée.

Les macros `if_`, `endif_` et `else_` sont génériques et indépendantes du langage du compilateur du processeur cible, elles appellent les macros `basicIf_`, `basicEndif_` et `basicElse_` qui sont spécifiques au langage du compilateur. Les macros génériques permettent de mettre en forme le code généré (indentation) et de vérifier le balancement des macros.

7.2.3 Séquences de communications

Comme la séquence de calcul d'un opérateur, les séquences de communication des communicateurs sont composées de trois parties : initialisation, itération, et finalisation. La partie itérative est codée par une liste de macros de communication, de synchronisation, et de contrôle structuré de répétition ou d'exécution conditionnelle.

7.2.3.1 Structure

Le début de chaque séquence de communication d'un communicateur com est marqué par une une macro `thread_(type_comm, nom_comm, <ListeNomsProcesseursConnectésComm>)` (à chacune de ces macros correspond une macro `spawn_thread_(nom)` placée dans la phase d'initialisation de la séquence de calculs), et s'achève par la macro `endthread_`.

Chacune des séquences de communication est ensuite structurée de la façon suivante :

1. une macro **type_comm_ini** correspondant à l'initialisation du communicateur (par exemple pour la mise en place de vecteurs d'interruptions, le démasquage des interruptions, etc),

2. une ou plusieurs macros `loadFrom_(nomProcr, listeCommunicateur)` et/ou `loadOnto_(nomComm, listeCommunicateur)`, de chargement arborescent des programmes, nous les étudierons en détail dans la section 7.2.3.4, p. 165,

3. l'initialisation des sémaphores tampon vide (les `Pre_empty` des `Suc_empty` car il faut les débloquer pour la première itération),

4. le début de la boucle de contrôle itératif est marqué par `loop_`,

5. la phase itérative comprend des macros d'émission ou de réception, chacune précédée et/ou suivie de macros de synchronisation,

6. la fin de la boucle de contrôle itératif est marquée par `endloop_`,

7. le code de finalisation (ou de désactivation) du communicateur **type_comm_end** pour programmer le communicateur dans un état inactif en fin d'exécution du programme distribué.

Remarque 41 *Les macros de gestion des communicateurs sont suffixées par le nom du type du communicateur car il peut y avoir plusieurs types de communicateurs dans une architecture hétérogène.*

7.2.3.2 Synchronisations

Nous avons vu que les communicateurs des processeurs actuels ne sont pas complètement autonomes, ils correspondent à des canaux de DMA qui requièrent quelques cycles du séquenceur d'instructions de l'opérateur. Ce partage nécessite un arbitrage et une gestion simultanée du contexte (état d'avancement) de la séquence de calculs et des contextes des séquences de communication. Comme l'expérience montre que les communications sont souvent critiques, il faut minimiser leur latence, et donc arbitrer l'allocation du séquenceur d'instructions en priorité au service des communications. Cependant, cette allocation prioritaire doit se limiter au strict nécessaire : pendant qu'une opération de communication attend la fin d'une opération de calcul pour démarrer, il faut que le séquenceur d'instructions soit alloué à la séquence de calculs pour avoir une opportunité d'exécuter la macro-opération de calcul dont la fin est attendue ; de même, pendant que l'opérateur de communication séquence les transferts de données, le séquenceur d'instruction doit être alloué à la séquence de calculs afin de tirer parti au mieux du parallélisme disponible entre opérateurs et communicateurs (canal DMA). Cette allocation ne doit donc avoir lieu que dès qu'un tampon mémoire est prêt à être transféré et dès que le communicateur est disponible pour le transfert (et juste pour la durée nécessaire à son relancement). Ces deux événements sont générés par deux sources différentes : le premier par le séquenceur d'instructions et le second par le communicateur. En général ces deux événements ne sont donc pas simultanés et l'ordre de leurs occurrences peut être quelconque, c'est pourquoi nous les avons synchronisé par l'insertion de sommets `Pre` et `Suc` (Cf. § 6.2.2).

Remarque 42 *Traditionnellement, le parallélisme et la synchronisation, entre calculs et communications, passent par le découpage des données en paquets recopiés dans des tampons mémoire alloués dynamiquement et organisés en files d'attente gérées sous interruption de fin de communication de paquet. Notre méthode évite les coûts d'allocation et de recopie en partageant, en exclusion mutuelle, les tampons mémoire de données, entre une séquence composée de macro-opérations de calcul et de synchronisation et une autre séquence composée de macro-opérations de communication et de synchronisation.*

Comme par ailleurs les liaisons de communication sont généralement des ressources "lentes", il faut les utiliser au maximum, donc l'arbitrage doit allouer le séquenceur d'instructions à une séquence de communication prioritairement par rapport à la séquence d'opérations de calcul. Par contre, il ne faut pas qu'une séquence de communication "gaspille" inutilement le temps du séquenceur d'instructions. Or, tous les communicateurs sont conçus au niveau matériel pour être capables de requérir (émission d'une interruption, Cf. §1.1.2.3.1, p. 11) le séquenceur d'instructions en fin de transfert. Cette réquisition déclenche une sauvegarde automatique du contexte du séquenceur d'instructions (au moins son pointeur d'instructions) et initialise celui-ci pour exécuter le "programme d'interruption" associé à l'opérateur de communication (le mécanisme de sélection du programme d'interruption fonction de la source de la requête varie d'un processeur à l'autre). Grâce à cela, il n'est pas nécessaire d'allouer le séquenceur d'instructions à une séquence de communication pendant ces périodes d'attente soit d'une fin de transfert, soit d'une synchronisation. Ainsi la séquence de calcul sera interrompue juste pendant le temps nécessaire soit à la programmation des registres de configuration du communicateur, soit à l'exécution des opérations de synchronisation entre les séquences de communication et la séquence de calcul.

Dans les deux cas, les paramètres de l'opération de communication (identification du communicateur, adresse et taille de la zone mémoire à transférer, ou bien adresse du sémaphore de synchronisation) sont déterminés lors de la compilation, donc inclus sous forme de constantes littérales dans le code de l'opération de communication. Ainsi, le contexte de la séquence de communication, à sauvegarder pendant ces périodes d'attente, est limité à l'adresse de la première instruction de l'opération de communication suivant celle en attente de complétion. Par ailleurs, le contexte de la séquence de calcul, à sauvegarder pendant ces interruptions par les séquences de communications, est limité aux quelques registres utilisés pendant les opérations

de communication par les opérations de synchronisation et de programmation des communicateurs, donc pendant la durée de l'interruption. Il suffit de sauvegarder ces quelques registres sur la pile d'appel de sous-programmes de la séquence de calcul. Les coûts de ces changements de contexte réduits (de l'ordre d'une demi-douzaine d'instructions par exemple pour un TMS320C40) sont considérablement inférieurs à ceux des exécutifs multitâches classiques qui requièrent une sauvegarde et une restitution de tous les registres du processeurs (puisque à la compilation de l'OS, rien ne permet de savoir lesquels seront utilisés) lors d'un changement de contexte entre deux tâches (de l'ordre d'une centaine d'instructions pour un TMS320C40).

La dissymétrie d'arbitrage du séquenceur entre la séquence de calcul et celle de communication implique une dissymétrie de l'implantation des macros de synchronisation Pre et Suc pour chaque séquence. En effet, une étape d'attente Suc , si elle peut attendre activement côté calcul (c'est-à-dire en utilisant le séquenceur d'instructions, qui sera requis sur interruption par un événement de fin d'opération de communication), doit côté communication attendre passivement (c'est-à-dire sans utiliser le séquenceur d'instructions, sinon la séquence de calcul n'aura jamais l'opportunité d'exécuter le Pre correspondant). Pour cette raison, on nommera par la suite :

Suc0_ les macros d'attente côté calcul, (0 pour "basse" priorité d'arbitage),

Suc1_ les macros d'attente côté communication (1 pour "haute" priorité d'arbitrage).

On nommera Pre0_ et Pre1_ les macros de synchronisation correspondant respectivement aux Suc0_ et Suc1_ (Cf figure 7.1).

7.2.3.2.1 Pre0_/Suc0_ Si la séquence de communication est exécutée sous interruption de la séquence de calcul, Suc0_ côté calcul peut attendre en scrutant activement l'état de son sémaphore car l'opérateur n'a rien de mieux à faire en attendant l'interruption de fin de transfert. Cette interruption requiert le séquenceur d'instructions pour exécuter, dans la séquence de communication, l'opération Pre0_ attendue par le Suc0_. Lorsque la séquence de communication exécute l'instruction de retour d'interruption, la séquence de calcul reprend l'exécution interrompue de Suc0_.

Exemple 7.2.9 *Voici des exemples de définitions des macros de précédence avec attente active* Pre0_/Suc0_ *générant du pseudo assembleur, on suppose que les sémaphores ont été alloués dans un tableau d'entier par la macro* sémaphores_ *(Cf. § 7.2.1.1).*

```
define('Pre0_',     '
sem_[$1]=1;    /* libére le sémaphore $1 */')

define('Suc0_',     '
while(sem_[$1]==0);      /* attend que le semaphore $1 soit libéré par 'Pre0_($1)'    */
sem_[$1]=0;    /* rebloque le semaphore $1 pour la prochaine itération   */') %$
```

La boucle de la macro Suc0_ *réalise bien une attente active qui sera débloquée par l'exécution du* Pre0_ *correspondant exécuté sous interruption après la fin d'une opération de communication. Voici le code qui sera généré :*

```
semaphores_(un_full,      un_empty,    deux_full,    deux_empty)
-> #define   un_full   3
   #define   un_empty   2
   #define   deux_full   1
   #define   deux_empty   0
   volatile   int  sem_[4]={0}    /* les sem.  sont  initialement    bloqués*/  ;

Pre0_(deux_full)
```

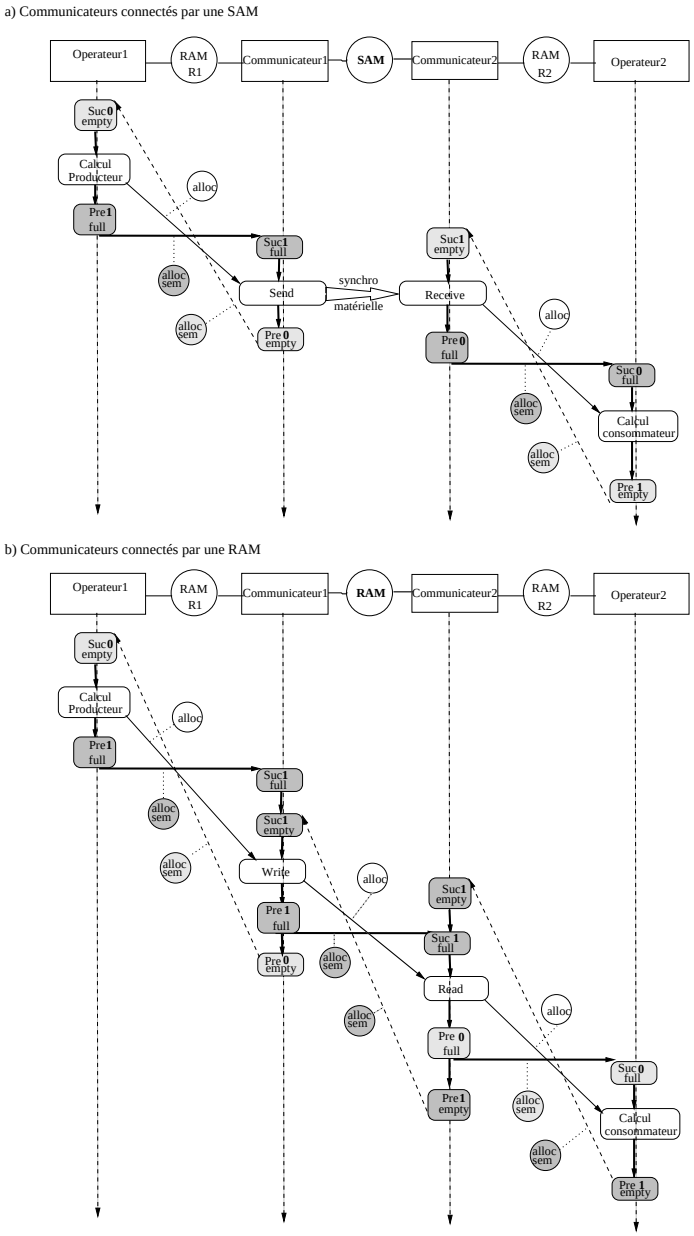


FIG. 7.1: *Synchronisations entre opérateur et communicateur*

```
->sem_[deux_full]=1;        /* lib ére le semaphore deux_full */

Suc0_(deux_full)
->while(sem_[deux_full]==0);        /* attend que le semaphore deux_full soit lib ér é
                                    par Pre0_(deux_full) */
  sem_[deux_full]=0;        /* rebloque le semaphore deux_full pour la prochaine it ération*/
```

7.2.3.2.2 Pre1_/Suc1_ Ce cas est plus complexe. Comme la séquence de communication est exécutée sur interruption de la séquence de calcul, `Suc1 _` côté communication ne peut attendre en scrutant activement

l'état de son sémaphore, elle doit rendre (par retour d'interruption) le séquenceur à la séquence de calcul.

Dans le cas où la séquence de calcul exécute un Pre1_ avant que le Suc1_ correspondant soit exécuté par la séquence de communication, il suffit que Pre1_ change l'état du sémaphore pour signaler son passage et que Suc1_ reconnaisse ensuite cet état pour continuer sans s'arrêter après avoir remis le sémaphore dans son état initial.

Par contre, dans le cas où la séquence de communication exécute un Suc1_ avant que la séquence de calcul ait exécuté le Pre1_ correspondant, comme la séquence de communication est exécutée en interruption de la séquence de calcul, on ne peut se permettre une attente active comme pour les précédences Pre0_/Suc0_, donc Suc1_ doit sauvegarder le contexte de la séquence de communication et restituer celui de la séquence de calcul interrompue. Plus tard, l'exécution du Pre1_ correspondant "auto"-interrompt la séquence de calcul en sauvegardant son contexte et en restituant celui de la séquence de communication. Comme les contextes des séquences sont alors limités à leur pointeur d'instruction et comme celui de la séquence de communication est connu à la compilation (l'adresse de l'instruction après Suc1_), pour changer de contexte il suffit d'une instruction d'appel de sous-programme dans Pre1_ et d'une instruction de retour de sous-programme dans Suc1_, dont l'exécution doit être conditionnée par l'état du sémaphore après sa modification.

Remarque 43 *Il est nécessaire que la lecture, le test et la modification du sémaphore soient effectués en exclusion mutuelle entre* Pre1_ *et* Suc1_*, donc interruptions inhibées car l'exécution de* Suc1_ *peut être déclenchée par une interruption générée par le DMA en fin de transfert.*

Exemple 7.2.10 *Voici les exemples de définitions de macros de précédence avec attente inactive, générant du pseudo-assembleur :*

```
define('Pre1_',     `
if(  (sem_[$1]=sem[$1]^1)==0       )
 call  Suc_$1_;   /* le label Suc_$1_  est défini  par  la macro  'Suc1_($1)'*/')

define('Suc1_',     `
if((sem_[$1]=sem[$1]^1=1)==1)        /* modifie  le semaphore  $1, si pas eu passage)   */
            returnFromCall;         /* return,  sinon continuer  seq. comm  */
Suc_$1_:   /* label  pour  le call  du Pre1  */
')
```

Dans le cas où la séquence de calcul est en avance sur la séquence de communication, Pre1 *est exécutée avant* Pre1 *.* Pre1 *marque son passage en complémentant la valeur du sémaphore (passage de 0 à 1). Quand* Suc1 *est exécuté, elle détecte que* Pre1 *a déjà été exécutée, les opérations qui la suivent sont exécutées.*

Dans le cas où la séquence de communication est en avance sur la séquence de calcul, Suc1 *est exécutée avant* Pre1 *. Elle indique son passage en changeant l'état du sémaphore (passage de 0 à 1) puis suspend la séquence de communication exécutée sous interruption en effectuant un retour d'interruption. Quand* Pre1 *est exécutée, elle détecte que* Suc1 *a déjà été exécutée ce qui a conduit à la suspension de la séquence de communication.* Pre1 *relance donc cette séquence de communication en effectuant un saut à l'instruction qui suit le* Pre1 *.*

Comme la condition d'exécution du call *de* Pre1_ *est l'opposée de celle du* return *du* Suc1_*, il est possible de simplifier les deux macros :*

```
define('Pre1_',     `
call  Suc_$1_;   /* le label  Suc_$1_  est défini  par  la macro  'Suc1_($1)'*/')

define('Suc1_',     `
```

```
if((sem_[$1]^=1)!=0)      returnFromCall;     /* modifie   semaphore   $1 */
Suc_$1_:   /* label  pour  le  call  du  Pre1  */
')
```

Voici le code généré en utilisant ces macros :

```
semaphores_(un_empty,       un_full)
->#define   un_empty   1
  #define   un_full   0
  volatile   int  sem_[2]={0};

Pre1_(un_empty)
->call   Suc_un_empty_;     /* le  label  Suc_un_empty_    est  défini
                             par  la  macro  Suc1_(un_empty)*/

Suc1_(un_empty)
->if((sem_[un_empty]^=1)!=0)        returnFromCall;     /* modifie   semaphore   un_empty   */
Suc_un_empty_:     /* label  pour  le  call  du  Pre1  */
```

7.2.3.3 Opérations de communication

Transfert RAM - SAM La macro `send_(nom_tampon, type_proc_emeteur, nom_proc_emetteur, liste_nom_proc_dest)` implante les opérations de communication de O''_{SEND}. Elle effectue le transfert du tampon de nom "nom_tampon" entre la RAM du processeur "nom_proc_emetteur" de type "type_proc_emeteur" et la SAM connectée au communicateur qui exécute cette macro. Ce dernier est connecté à un processeur de la liste "liste_nom_proc_dest".

Inversement la macro `recv_(nom_tampon), type_proc_emeteur, nom_proc_emetteur, liste_nom_proc_dest)` implante les opérations de communication de $O''_{RECEIVE}$. Elle effectue le transfert du tampon de nom "nom_tampon" entre la RAM et les SAM connectées au communicateur qui exécute cette macro.

Enfin, utilisée quand la mémoire SAM supporte matériellement le broadcast, la macro `sync(type_données, nombre_données,nom_proc_emetteur, nom_proc_dest)` implante les opérations de communication de O''_{sync}. Elle permet de synchroniser la séquence de transfert avec les autres séquences de transferts exécutées par les communicateurs connectés aux même mémoires.

Remarque 44 *Dans une architecture hétérogène, chaque type de données peut avoir des représentations mémoire différentes pour les différents types de composants de l'architecture. Il est donc nécessaire de définir des macros de transferts inter-mémoire différentes pour chaque type de donnée, afin de supporter lors du transfert les conversions entre les différentes représentations mémoire. Par contre, comme tous les éléments d'un tableau ont la même taille et sont contigus en mémoire, il suffit d'une seule macro de transfert inter-mémoire pour tous les types tableau d'un même type scalaire, cette macro prend en argument le nombre total d'éléments à transférer (un seul pour le type scalaire lui-même, sinon le produit des dimensions du tableau).*

Transfert RAM - RAM La macro `write(nom_tampon_lu, nom_tampon_ecrit)` (resp. `read(nom_tampon nom_tampon_ecrit)`) implante les opérations de communication de O''_{write} (resp. O''_{read}). Ces deux macros transférent les données du tampon "tampon_lu" dans le tampon "nom_tampon_ecrit".

Dans le cas d'une macro `write` le premier tampon est alloué dans une RAM partagée par le communicateur qui exécute l'opération et un opérateur. Le second tampon est alloué dans une RAM partagée par des

communicateurs. Dans le cas d'une macro read c'est le contraire.

Exemple 7.2.11 *En pratique, quand le communicateur est un canal DMA, le code des opérations de communication est très similaire que ce soit pour un transfert RAM/SAM ou RAM/RAM car la différence de gestion des mémoires est encapsulée dans le DMA. Ces opérations se réduisent à programmer les registres d'un canal DMA, à sauvegarder l'adresse de l'instruction qui suit l'appel de la macro de transfert, et à restaurer le contexte de la séquence de calcul. Le programme d'interruption de fin de transfert doit sauvegarder le contexte de la séquence de calcul et reprendre l'exécution de la séquence de communication à l'adresse sauvegardée.*

Dans le cas d'architecture hétérogène, les communicateurs peuvent être de types différents, nous avons donc choisi de nommer les fonctions de gestion de communicateur en les préfixant par le nom de type de communicateur. Pour cela on commencera toujours par substituer les macros génériques send_, recv_, sync_, read_, write_ *par des macros spécifiques au type de communicateur pour lesquels elles sont utilisées. Comme le type de communicateur est identifié dans la macro* thread_(type_comm, nom_comm, <ListeNomsProcesseurConnectésComm>) *qui précède la séquence de communication, la définition suivante permet de faire la substitution automatiquement des macros de communications :*

La macro thread_ *extrait, entre autre, le type du communicateur et crée une macro* commType *:*

```
define('thread_', 'dnl
pushdef('commType_',$1)dnl
')
```

Ensuite, les macro-opérations de communication peuvent être substituées automatiquement par des macros préfixées par le type de communicateur (dans l'exécutif que l'on génère, nous profitons de ces macros pour implanter d'autres fonctionnalités comme par exemple la vérification de l'existence du tampon à transmettre et l'existence d'une définition pour la macro obtenue), voici la macro qui substitue les macros send *par sa version spécifique au type de communicateur :*

```
define('send_','
commType_()_send_($@)')
```

Ainsi, la séquence de communication suivante sera substituée par :

```
thread_(DMAC40, bus1, P1)
 send_(tampon1, C, P1, root)
->DMAC40_send_(tampon1,C,P1,root)
```

Exemple 7.2.12 *Voici un exemple de définition de macros générant du pseudo-assembleur pour un communicateur de type C40 ("DMAC40") de transfert entre une RAM (partagée avec le processeur) et une SAM point-à-point. Pour programmer chaque canal du DMA, nous utilisons une structure de données que nous définissons lors de la phase d'initialisation du communicateur :*

```
typedef struct{
  int    control;      /* sens du transfert: START_DMA_OUTPUT ou START_DMA_INPUT */
  int    counter;      /* quantité de données a transferer */
  char  *address;      /* adresse de début de la zone mémoire à transferer */
  void(*suspend)();    /* addresse des instructions interrompues pendant le trans-
fert */
} DMAchannels[NUMBER_OF_DMA_CHANNELS];
#define DMAchannel (DMAchannels)BASE_ADDRESS_OF_MEMORY_MAPPED_DMA_REGISTERS

define('DMAC40_ini_',      'dnl initialise le communicateur
```

```
DMA$1_interrupt:        /* exemple type d'une routine d'interruption de fin de trans-
fert */
  saveRegistersUsedDuringInterrupt( );
  call(DMAchannel[$1].suspend);        /* execution des instruction interrompues */
  restoreSavedRegisters();
  returnFromInterrupt;
DMA$1_sequence:          /* point d'entr é de la séquence de communication du DMA$1 */
  enableDMAinterrupt($1);
```

Maintenant, nous pouvons utiliser cette structure dans les macros send, recv et sync *pour programmer le DMA :*

```
%%# DMAC40_recv_(1bufferName,        2senderType,3senderName        {, receiverNames})

define('DMAC40_send_',       'dnl par $1 envoie $2 donn ées de l'adresse $3
  DMAchannel[$1].counter       = $2*sizeof(int);     /* taille transfert */
  DMAchannel[$1].address       = (char*)$3;    /* adresse source */
  DMAchannel[$1].suspend       = resume$3;     /* Label déclar é ci-dessous */
  DMAchannel[$1].control       = START_DMA_OUTPUT;        /* active DMA */
  returnFromCall;     /* d'une IT DMA$1 ou d'un 'Pre1_' */
resume$3:    /* adresse d'instruction à reprendre par IT DMA$1 */')

define('DMAC40_rceive_',        'dnl par $1 recoit $2 donn ées de l'adresse $3
  DMAchannel[$1].counter       = $2*sizeof(int);     /* taille transfert */
  DMAchannel[$1].address       = (char*)$3;    /* adresse destination */
  DMAchannel[$1].suspend       = resume$3;     /* Label déclar é ci-dessous */
  DMAchannel[$1].control       = START_DMA_INPUT;         /* activate DMA */
  returnFromCall;     /* d'une IT DMA$1 ou d'un 'Pre1_' */
resume$3:    /* adresse d'instruction à reprendre par IT DMA$1 */')

define('DMAC40_end_',       'dnl termine sequence communication $1
  disableDMAinterrupt();       /* et autre DMA$1 finalisations */
  returnFromCall;     /* de l'IT DMA$1 ou d'un 'Pre1_' */')
```

Remarque 45 *Dans les cas où il n'y a pas de DMA (par exemple un micro-contrôleur pilotant une interface série du genre RS232) le séquencement des opérations de communication doit être supporté par le séquenceur d'instructions, donc en interrompant la séquence de calcul à chaque transfert. Les opérations de communications doivent alors sauvegarder les paramètres de la communication qu'elles reçoivent en arguments (adresse et taille de la zone mémoire à transférer) et armer un programme d'interruption qui, appelé à la fin de chaque micro-transfert, incrémente l'adresse courante, décrémente la taille restant à transférer et relance un micro-transfert, et ce jusqu'à ce que la taille restante devienne nulle, après quoi la séquence de communication est reprise à l'instruction suivant l'appel de la macro de transfert, comme dans le cas avec DMA. Ce type de comme est cependant très pénalisant étant donnée l'abscence de parallélisme entre calcul et communication que l'on modélise au moyen de la politique de l'arbitre d'accès à la mémoire partagée entre communicateur et opérateur.*

7.2.3.4 Chargement arborescent des programmes

Comme dans tout exécutif, il faut supporter le chargement initial des mémoires des processeurs. Habituellement, un seul processeur "hôte" est équipé de mémoire de masse non volatile, disque ou EPROM, contenant les programmes à charger sur les autres processeurs. Nous ne nous préoccuperons pas ici du chargement du processeur hôte, mais uniquement de celui des autres processeurs effectué à partir du processeur

hôte. On supposera que l'hôte démarre ("boot") à partir de sa mémoire de masse, charge éventuellement un système d'exploitation pour gérer des entrées-sorties standard ("stdio" : clavier, écran, disque), et au besoin requiert la saisie par un utilisateur d'une commande de lancement pour terminer son propre chargement. C'est à partir du point d'entrée du `main` de l'hôte que l'on s'intéresse ici à générer l'exécutif. On supposera également que le démarrage ("reset") de chacun des autres processeurs est commandé par le processeur hôte. Il faut donc choisir un arbre de couverture du graphe d'interconnexion des processeurs, ayant pour racine le processeur hôte (nous choisirons un nom explicite,"root" pour différencier ce processeur particulier).

Si l'hôte a un accès direct à la mémoire programme de ses descendants dans l'arbre, alors il chargera leur programme avant de commander leur démarrage. Sinon, on supposera que chaque processeur a une mémoire non volatile de démarrage contenant un programme de chargement ("boot loader") à travers une liaison physique de communication, auquel cas l'hôte commandera le démarrage de tous les processeurs et leur transmettra leur programme, au besoin par l'entremise des processeurs intermédiaires dans l'arbre.

Ainsi, la phase initiale du programme de l'hôte consiste d'abord à extraire de sa mémoire de masse le programme de chaque processeur. Ensuite il doit transmettre chacun de ses programmes au processeur qui lui correspond, au besoin par l'entremise des processeurs intermédiaires dans l'arbre. La phase initiale des séquences de communication (Cf. § 7.2.3) de chacun des autres processeurs consiste à recevoir de son ascendant dans l'arbre et à transmettre à chacun de ses descendants, le programme qui lui est destiné. Les processeurs feuilles de l'arbre n'ont donc rien à faire pendant cette phase initiale puisqu'ils n'ont pas de descendants dans l'arbre de couverture du graphe de processeurs. Le chargement des processeurs selon cette arborescence est codée, dans la phase d'initialisation de la séquence de communication de chaque processeur, par des macros désignant, pour l'ascendant et pour chaque descendant du processeur, le communicateur qui l'y connecte.

- Pour la RAM ou la SAM connectée au processeur ascendant dans l'arbre de couverture :
  `loadFrom _(nomProcr, listeCommunicateurs)` où

  nomProcr est le nom du processeur ascendant dans l'arbre de couverture,

  listeCommunicateurs est la liste des mémoires connectées aux processeurs descendants

- Pour chaque communicateur connecté à un ou plusieurs processeurs descendants :
  `loadOnto _(nomCommunicateur, listeNomsProcr)`, où

  nomCommunicateur est vide pour le processeur *root*, sinon c'est le nom du communicateur connecté au processeur ascendant

  listeNomsProcr représente la liste des processeurs descendants.

Exemple 7.2.13 *Prenons l'exemple d'architecture hétérogène de la figure suivante (7.2), le nom de chaque processeur (ici en pointillés) sera identique à celui de l'opérateur qu'il contient (ce dernier étant unique). Chaque opérateur est ici connecté à une RAM elle même connectée à un ou plusieurs communicateurs. Le chargement arborescent des programmes consiste ici à charger les RAM r1 à r6. Soit $Oprroot$ le processeur hôte. Étant donnée la topologie de cette architecture, les chargements de tous les autres processeurs se font par l'intermédiaire du communicateur $C1$ de $Oprroot$. Le premier processeur à recevoir les programmes de $Oprroot$ étant $Opr2$, on trouvera dans la phase d'initialisation de $C1$ de $Oprroot$ une macro de chargement* `loadOnto _(, Opr2)` *(le premier argument est vide car $Oprroot$ est connecté à l'hote).*

Comme $Opr2$ doit recevoir les programmes en provenance de $Oprroot$ à travers son communicateur $C2$ pour les transmettre aux processeurs $Opr3$ et $Opr4$ à travers ses communicateurs $C3$ et $C4$, on trouve la macro `loadFrom _(Opr root, C3, C4)` *dans la phase d'initialisation de $C1$ de $Opr2$. Pour les même raisons on trouve respectivement les macros* `loadOnto _(C2, Opr3)` *et* `loadOnto _(C2, Opr4)` *dans les phases d'initialisation des communicateurs $C3$ et $C4$.*

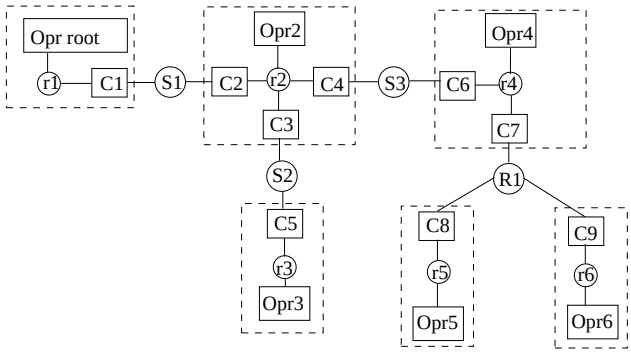


FIG. 7.2: *Chargement arborescent d'une architecture*

$Opr3$ reçoit son programme en provenance de $Opr2$ et n'en transmet à aucun autre puisque c'est une feuille de l'arbre de couverture, on trouve donc la macro `loadFrom _(Opr2, )` *dans la phase d'initialisation du communicateur $C5$ de $Opr3$.*

$Opr4$ reçoit de $Opr3$ les programmes des processeurs $Opr5$ et $Opr6$ par son communicateur $C6$. Comme ces deux opérateurs sont connectés à la même RAM partagée chargée par le communicateur $C7$ de $Opr4$, on trouve la macro `loadFrom _(Opr3, C7)` *dans la phase d'initialisation de $C6$ et la macro* `loadOnto _(C6, Opr5, Opr6)` *dans la phase d'initialisation de $C7$.*

Comme $Opr5$ et $Opr6$ sont racines de l'arbre de couverture du graphe d'architecture et qu'ils sont chargés par le même processeur, on trouve les mêmes macros `loadFrom _(Opr4, )` *dans la phase d'initialisation de leur communicateur respectif $C5$ et $C6$.*

7.2.4 Chronométrage

Comme nous l'avons évoqué dans le chapitre consacré à l'optimisation, il est possible, si chaque processeur possède une horloge temps réel (timer), d'ajouter des opérations de chronométrage pour :

- caractériser, pour chaque nouvelle architecture matérielle, les opérations de synchronisation et les opérations de communication du noyau générique d'exécutif,
- caractériser chaque nouvelle macro-opération utilisée dans de nouveaux algorithmes,
- vérifier pour chaque implantation d'un algorithme sur une architecture, que les performances prédites et calculées à partir des caractéristiques mesurées, correspondent à la réalité.

Le chronométrage consiste à insérer des opérations de chronométrage entre les opérations à caractériser. Chaque opération insérée effectue une mesure dont la valeur est stockée dans un tampon circulaire. Il est important de souligner que ces opérations ne sont pas nécessaires au bon fonctionnement de l'application, elles ne sont utilisées que dans la phase de développement de l'application. Le chronométrage d'applications temps réel embarquées distribuées présente plusieurs difficultés :

- comme chaque processeur possède sa propre horloge, il faut considérer que la notion du temps est locale à chaque processeur et donc mesurer (par l'intermédiaire de communications inter-processeurs qui prennent elle-même du temps qu'il faut caractériser) les décalages entre horloges pour pouvoir reconstituer une notion globale du temps,

– les contraintes temps réel et d'embarquabilité incitent à minimiser les surcoûts de chronométrage, et donc à reporter en phase de finalisation, après la dernière itération temps réel, les surcoûts de mesures de décalages entre horloges et ceux de collecte des chronométrages, et pour limiter les surcoûts de mémorisation avant collecte, à ne conserver que les chronométrages des quelques dernières itérations.

il s'effectue en quatre phases:

1. **Phase d'initialisation :** l'allocation des tampons circulaires est faite par la macro `Chrono_(NbPoints)`, placée à la suite des macros d'allocation mémoire. L'argument qu'elle reçoit correspond au nombre de mesures effectuées lors d'une itération. Chaque mesure comprend deux entiers, l'un étant une étiquette identifiant le point de mesure (entre deux opérations de la séquence de calcul ou de la séquence de communication), l'autre étant une date lue sur l'horloge temps réel locale du processeur. L'initialisation des tampons circulaires de stockage des mesures est effectuée par la macro `Chrono_ini_` placée à la fin de la phase d'initialisation de la séquence de calculs.

2. **Phase de mesure des dates :** La mesure des dates, indépendante sur chaque processeur, est effectuée pendant le déroulement du programme par des macros de chronométrage `Chrono_lap_(Etiquette, NomTampon)` insérées entre les opérations de la séquence de calcul et entre les opérations de la séquence de communication. Chaque point de mesure est identifié par une étiquette différente passée en argument de la macro. Cette étiquette est enregistrée, avec la date mesurée sur l'horloge temps-réel du processeur, dans un tampon circulaire dimensionné de manière à retenir les mesures des quelques dernières itérations (réactions) du programme.

   Cette seconde phase de mesure des dates pose un problème délicat : il faut définir une condition d'arrêt pour passer de la seconde à la troisième phase. La difficulté consiste à arrêter tous les processeurs lors de la même itération afin qu'ils soient ensuite tous synchronisés pour les deux dernières phases de chronométrage. Pour les programmes sans itération, comme c'est le cas dans certains tests, il n'y a aucun problème. Pour les programmes avec itération, nous fixons, à la compilation, un nombre d'itérations commun à tous les processeurs en conditionnant les macros de contrôle itératif `loop_` et `endloop_` par l'existence de la macro `NBITERATIONS` comme cela a été présenté dans le paragraphe 7.2.2. Ainsi, si la macro `NBITERATIONS` est définie, les macros `loop_` et `endloop_` génèrent, pour chaque séquence de calcul ou de communication, un code de contrôle imposant un nombre d'itérations fixé par la valeur de la macro `NBITERATIONS`.

   Remarque 46 *La lecture d'une date sur l'horloge temps réel, ainsi que son stockage dans le tampon circulaire, ne sont pas gratuits : la durée d'exécution d'une macro opération de chronométrage doit être caractérisée (mesurée pour chaque type de processeur) pour pouvoir être retranchée de chaque intervalle entre deux dates si l'on désire avoir des mesures précises de durées des macro-opérations. Pour mesurer la durée d'exécution d'une macro-opération de chronométrage, il suffit d'en exécuter deux contigües, ou mieux plusieurs pour pouvoir calculer une durée moyenne et se faire un idée des variations s'il y en a.*

3. **Mesure des décalages entre horloges :** une troisième phase est effectuée après la fin des calculs (phase de finalisation), par la première partie de la macro `Chrono_end_` générée en fin de phase de finalisation de chaque séquence de calcul. Cette phase consiste à mesurer les décalages entre les horloges des processeurs, afin de rendre comparables entre elles les dates mesurées sur des processeurs différents. Le processeur hôte (à la racine de l'arbre de couverture du graphe d'interconnection des processeurs qui est utilisé pour le chargement arborescent des programmes, voir section précédente, puis récursivement chacun de ses descendants), effectue une communication aller-retour avec chacun

de ses descendants dans l'arbre. En prenant la précaution pendant cette phase d'éviter toute interférence entre calculs et communications, on peut considérer que la date de renvoi du message, mesurée par le descendant, correspond à la moyenne des dates d'émission et de réception du message, mesurée par l'ascendant. Comme les branches de l'arbre sont séparées, ce processus de mesure des décalages entre les horloges des processeurs peut être effectué en parallèle dans chaque sous-arbre sans qu'il y ait d'interférence qui puisse perturber la précision de la mesure.

Soient :

- t le temps de l'horloge locale
- t_1 la date d'émission du message aller
- t_2 la date de réception du message retour
- t' le temps de l'horloge du processeur parent
- τ' la date de réception/renvoi sur le processeur parent

La date $\tau = (t_1 + t_2)/2$ correspond à τ', ce qui donne la relation $t - \tau = t' - \tau'$ d'où l'on tire $t' = t + (\tau' - \tau)$. Il faut donc ajouter $\Delta t = \tau' - \tau$ aux dates mesurées sur l'horloge locale pour les rendre comparables à celles mesurées sur l'horloge du processeur parent dans l'arbre de couverture du graphe des processeurs.

Remarque 47 *En pratique, les dates étant des entiers non signés, il faut utiliser l'arithmétique modulo*[1] *et calculer* $\tau = t_1 + (t_2 - t_1)/2$*, ce qui donne* $\Delta t = \tau' - (t_1 + (t_2 - t_1)/2)$*, ou encore, pour la commodité de l'implantation du calcul,* $\Delta t = (t_2 - t_1)/2 - (t_1 - \tau')$.

La mesure des décallages s'effectue grâce à deux macros complémentaires. Côté parent, la macro `ChronoSync_` envoie un premier message pour signaler que le processeur est prêt à faire la mesure, puis attend de recevoir t_1 pour mesurer τ' et renvoyer immédiatement $t_1 - \tau'$. Côté descendant, la macro `ChronoDiff_` attend le message "prêt" de son parent, puis mesure t_1 qu'il envoie immédiatement, puis attend de recevoir $t_1 - \tau'$ pour mesurer t_2 qu'il retranche alors à t_1 et divise le résultat par deux et y retranche $t_1 - \tau'$ pour obtenir Δt qu'il retourne en résultat. Ces deux macro sont de préférence implantées en assembleur pour leur donner le maximum d'efficacité afin que la précision des mesures soit la meilleure possible.

Il est très important, avant d'effectuer les mesures de décalage entre les horloges, de s'assurer que toutes les séquences de communication sont achevées afin de ne pas engendrer d'interférences. Pour cela, lorsque l'on génère un exécutif avec chronométrage, nous ajoutons autant de macro `wait_end_thread_(NomComm)` qu'il y a de communicateurs, avant la macro `Chrono_end_()` de la séquence de calcul.

4. **Collecte des chronométrages :** la quatrième et dernière phase correspond à la seconde partie de la macro `Chrono_end_` générée en fin de phase de finalisation de chaque séquence de calcul. Cette phase consiste à collecter les résultats des chronométrages effectués en seconde phase sur chaque processeur. Les mesures sont collectées en remontant l'arbre de couverture du graphe d'interconnexion

1. car le bit de poids fort du r´esultat de l'addition $t_1 + t_2$ est perdu et ne peut ˆetre correctement restitu´e par la division par deux, sign´ee par d´efaut, alors que la dur´ee $t_2 - t_1$ est correctement sign´ee (sauf si sa magnitude est sup´erieure `a la moiti´e de la dynamique du compteur de l'horloge, soit environ 200 secondes pour un timer 32 bits `a 10 MHz comme celui du TMS320C40, cas que nous consid´erons exclus), donc la division par deux fournit un r´esultat correct qui peut ˆetre additionn´e `a $t_1$ sans probl`eme en utilisant l'arithm´etique modulo.

des processeurs qui est utilisé pour le chargement arborescent des programmes. Chaque processeur envoie d'abord ses propres mesures à son ascendant dans l'arbre, sauf l'hôte, à la racine de l'arbre, qui n'a pas d'ascendant, et qui stocke les résultats dans sa mémoire de masse, d'abord les siens puis ceux reçus de ses descendants. Puis chaque processeur retransmet à son ascendant les résultats qu'il reçoit de ses descendants. Lorsqu'il n'a plus de mesures à transmettre à son ascendant, il lui envoie un message "vide" (contenant une mesure avec l'étiquette nulle réservée à cet effet) qui permet au processeur ascendant de passer à son descendant suivant. La collecte se termine lorsque l'hôte n'a plus de descendant suivant.

7.2.5 Ossature d'un fichier processeur

Nous présentons ici une synthèse du macro-exécutif qui doit être généré pour chaque processeur de l'architecture :

1. include (syndex.m4x)
2. processor _(typeProcr, nomPrcr, nom _appli) (marque le d´ebut de l'ex´ecutif)
3. **allocations (déclarations) :**
   - allocation des tampons :
     - RAMarea _ / alloc _(nom, type, taille) / endRAMarea_
   - allocation des s´emaphores :
     - RAMarea _ / semaphores _(nom _sem _1_empty, nom _sem _1_full,nom _sem _2 /endRAMarea _ etc)
4. **séquence de communication de nom C_i :**
   - thread _(type,nom, procr1, procr2 ...)
     - *comm_type*_shared (g´en´er´ee si elle existe) pour la partie commune du code des communications,
     - point d'entr´ee pour spawn _ (C_i),
     - *comm_type*_ini (g´en´er´ee si elle existe) pour initialiser le communicateur,
   - initialisation ´etat des s´emaphores "tampon vide"
     - Pre0 _(nom _sem _empty) et Pre1 _(nom _sem _empty) (autant que n´ecessaire)
   - boucle principale de communication :
     - loop _
     - code de la s´equence de communication :

(lire SAM)
```
Suc1 _(nom _sem _empty)
recv _(nom _tampon, TypeProcSend, nomProcSend, listenomsProcsRcv)
Pre0(ou1) _(nom _sem _full)
```

et/ou (´ecrire SAM)
```
Suc1 _(nom _sem _full)
send _(nom _tamponTypeProcSend, nomProcSend, listenomsProcsRcv)
Pre0(ou1) _(nom _sem _empty)
```

et/ou (synchro SAM)
```
(Multi-point)
Sync _(nom _tampon,TypeProcSend, nomProcSend, listenomsProcsRcv)
```

et/ou (´ecrire RAM)
```
Suc1 _(nom _sem _Ram1 _full)
Suc1 _(nom _sem _Ram2 _empty)
write _(nom _tamponLu, nom _tamponEcrit)
Pre0(ou1) _(nom _sem _Ram1 _empty)
Pre0(ou1) _(nom _sem _Ram2 _full)
```

et/ou (lire RAM)
```
Suc1 _(nom _sem _Ram1 _full)
Suc1 _(nom _sem _Ram2 _empty)
read _(nom _tamponLu, nom _tamponEcrit)
Pre0(ou1) _(nom _sem _Ram1 _empty)
Pre0(ou1) _(nom _sem _Ram2 _full)
```

     endloop _(fi n de la boucle)
   - endthread _(fi n de la s´equence de communication num´ero $_iC$
     - *comm_type*_end ex´ecute le code de fi nalisation de ce type de communicateur.
       point de sortie, fi n de thread
5. **autre séquence de communication** (autant de s´equences que de communicateurs connect´es) :
   - thread _(C_{i+1}, ,),

   ../..
   - endthread _,
6. **séquence de calcul :**
   - main _ (point d'entr´ee pour le loader)
     - initialisations (pile, confi guration m´emoire ...),

- lancement des s´equences de communications :
  - spawn_thread_(C_i),
  - spawn_thread_(C_{i+1}) etc
- appel des macros d'initialisation des entr´ees constantes, des m´emoires et des I/O,
- initialisation ´etat des s´emaphores
  - Pre1_(nom_sem_empty) etc
- boucle principale de calcul
  - loop_
  - s´equence d'op´eration de calculs et de synchronisations, exemple :

```
Suc0_(nom_sem_tampon_entrée_full)
Suc0_(nom_sem_tampon_sortie_empty)
Calcul_(entrée, sortie)
Pre1_(nom_sem_tampon_sortie_full)
Pre1_(nom_sem_tampon_entrée_empty)
```

  - endloop_ (fin de la boucle principale)
- endmain_ (fin de la s´equence ce calcul)

7. endprocessor_ (fin du code pour ce processeur)

7.3 Génération de macro-exécutif

Nous avons vu que le macro-exécutif de chaque processeur correspond à une séquence de macro, stockée dans un fichier. Nous venons d'étudier la structure et le nom des macros utilisés dans ces fichier. Maintenant nous allons donner les algorithmes qui permettent de construire ces fichiers par exploration d'un graphe d'exécution (Cf. figure 7.3). Le nom de ce fichier est construit par la concaténation du nom du processeur avec l'extension ".m4x" (pour m4 eXecutif).

Le fichier correspondant à chaque processeur p de l'architecture est créé en trois étapes : construction de la séquence d'allocation mémoire, construction des séquences de communications, construction de la séquence de calcul. Les deux premières macros du fichier d'un processeur sont toujours :

- include(syndex.m4x)
- processor_(type de processeur, nom de processeur, nom_application)

7.3.1 Allocation mémoire

- pour chaque RAM m_i connectée à l'opérateur du processeur p ($\forall m_i \in \mu(p)$) ou connectée à un communicateur de p ($\forall m_i \in \mu_{com}(c_k)/c_k \in S_{com,p}$)
  - générer une macro RAMarea_((m_i)) afin d'indiquer que les tampons dont la déclaration va suivre seront alloués dans cette zone mémoire,
    - ◇ ajouter une macro semaphores_()
    - ◇ pour chaque sommet allocation $a_j \in O''_{alloc}$ dont l'opération productrice est une opération de synchronisation Pre ($\forall a_j \in \Pi^{-1}_{mem}(a_j) = m_i/\gamma^{-1}(\alpha^{-1}(a_j)) \in O''_{alloc}$) :
      - ⋆ générer le nom associé à cette opération Pre (nom opération productrice concaténée avec la valeur du 1er élément du quadruplet associée à la dépendance de données synchronisée par le Pre) concaténé avec le mot ''_empty'' si le sommet Pre est de type empty, et "_full " sinon (les noms sont séparés par des virgules),

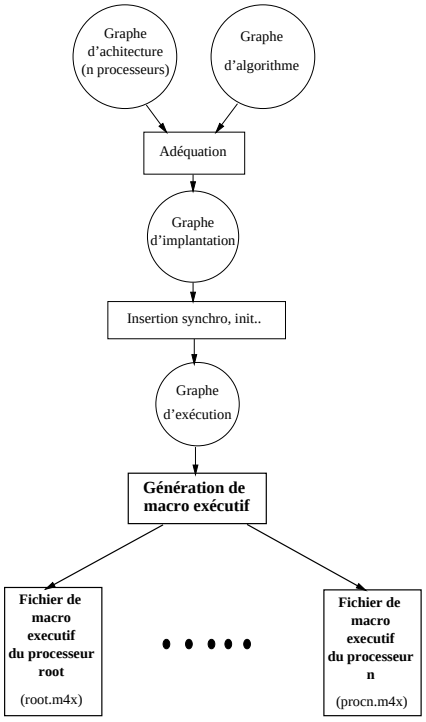


FIG. 7.3: *Génération de macro-exécutif à partir du graphe d'exécution*

◇ générer une parenthèse fermante ")",

◇ pour chaque sommet allocation $a_j \in O''_{alloc}$ distribué sur m_i dont l'opération productrice n'est pas une opération de synchronisation ($\forall a_j \in \Pi^{-1}_{mem}(a_j) = m_i / \gamma^{-1}(\alpha^{-1}(a_j)) \notin O''_{sync}$):

⋆ générer une macro `alloc _(type,nom,[tail le] )` (le type et la taille du tampon alloués sont tirés du quadruplet associé à la dépendance de données implantée par le sommet a_i, c'est à dire $type(\alpha^{-1}(a_i)$ et $q(\alpha^{-1}(a_i)$. Le nom est construit à partir du nom de l'opération de calcul ou d'entrée sortie productrice concaténé avec la valeur du premier élément du quadruplet associé à la dépendance de données implantée par a_i,

◇ pour chaque sommet alias $a_j \in O''_{alias}$ distribué sur m_i ($\forall a_j \in \Pi^{-1}(m_i) / a_j \in O''_{alias}$):

⋆ générer une macro `alias(nom _src,nom)` (où nom est le nom de ce sommet et nom_src celui de l'étiquette associée à ce sommet lors de l'optimisation mémoire)

∘ indiquer la fin de l'allocation dans cette zone par la macro `endRAMarea _`

Remarque 48 *on rappelle que chaque sémaphore est étiqueté par un nom qui est la concaténation du nom de l'opération productrice, d'un souligné, de l'indice du tampon dans la liste d'appel de la macro-opération (car il est unique et permet de construire un nom unique de sémaphore), d'un souligné du mot "full" pour définir un sémaphore de précédence tampon plein ou "empty" pour une précédence tampon vide, d'un souligné et enfin du nom du communicateur :*

```
NomOpn _IndiceDependance _full _NomComm ,

NomOpn _IndiceDependence _empty _NomComm .
```

7.3.2 Séquences de communication

Ensuite, pour chaque communicateur du processeur p, on explore la séquence d'opérations de communication qui lui a été associée pour générer les macros de transfert et de synchronisation.

- pour chaque communicateur $c_i \in S_{com,p}$,
  - marquer le début de la séquence par la macro thread_(type_comm, nom_comm, <ListeNomsProcesseursConnectésComm> et,
    1. ajouter les macros de téléchargement loadFrom_(nomProcr, listeCommunicateurs) et/ou loadDnto_(nomCommunicateur, listeNomsProcr) , selon l'arbre de couverture du graphe d'architecture (Cf. 7.2.3.4)
    2. ajouter la macro de la première opération exécutée par le communicateur : macro d'initialisation du communicateur comm_init_ insérée lors de la construction du graphe d'exécution (Cf. 6.2.1),
    3. libérer les synchronisations inter-itération (Cf. 3.2.4.4) en parcourant la séquence (i.e. en suivant l'ordre total $\bar{D}''_{c_i}$) associée au communicateur c_i à partir du sommet LOOP jusqu'à atteindre le sommet ENDLOOP :
       - ◇ pour chaque sommet Pre_empty ajouter une macro Pre_empty(*nom*) où *nom* est le nom associé au sommet Pre concaténé avec "_empty" (puisque ce sont nécessairement des sommets de type Pre_empty),
    4. marquer le début de la partie itérative en ajoutant la macro Loop_ et parcourir la partition associée au communicateur en suivant toujours les précédences à partir du sommet LOOP ,
       - ◇ pour chaque sommet o_j rencontré, ajouter la macro correspondante ($\forall o_j \in \Pi^{-1}(c_i)$) :
         - ⋆ pour chaque opération de communication $o_j \notin O_{sync}$ il faut générer une macro correspondant au type de sommet : (send_ pour un sommet de type SEND etc). Ces macros reçoivent au moins quatre arguments. Le premier est le nom de l'opération de calcul ou d'entrée-sortie productrice concaténée avec la valeur du premier quadruplet associé à la dépendance qui les connecte (afin d'identifier le tampon dans lequel les données seront lues ou écrites). Le second est le type du processeur exécutant l'opération de calcul ou d'entrée-sortie productrice et le troisième est le nom de ce processeur. Le quatrième argument est la liste de noms de processeurs destinataire si c'est un sommet qui écrit sur la mémoire partagée (SEND, WRITE), ou le nom du processeur si c'est un sommet de lecture (RECEIVE, READ, Synchro),
         - ⋆ pour chaque sommet de synchronisation Pre ou Suc $o_j \in O_{sync}$ la macro équivalente est générée. Son argument est le nom du tampon mémoire alloué pour le sémaphore, c'est à dire le nom associé à ce sommet lors de sa construction,
    5. lorsque le sommet ENDLOOP est atteint, ajouter la macro endloop_ pour marquer la fin de la partie itérative,
    6. ajouter les macros de chronométrages éventuelles qui correspondent au sommet successeur du sommet ENDLOOP ,
    7. ajouter le sommet de finalisation correspondant au sommet successeur dans la partition du communicateur,
  - marquer la fin de la séquence de communication correspondante à ce communicateur en ajoutant la macro endthread_

7.3.3 Séquence de calcul

Ensuite, il reste à générer la séquence de calcul de l'opérateur p_l :

- générer la macro `main _` pour marquer le début de cette séquence,
  - pour chaque communicateur c_i connectés à p_l,
    - ◇ générer une macro `spawn _thread _(c _i)`
  - pour chaque opération o_k ordonnancée sur p_l avant le sommet `Loop` ,
    - ◇ générer une macro de même nom que o_k, sans générer d'arguments puisque ce sont les opérations d'entrée-sortie d'initialisation,
  - générer la macro `loop _`
  - pour chaque opération o_k ordonnancée sur p_l avant le sommet `endloop` ,
    - ◇ si c'est une opération de calcul ou d'entrée-sortie ($o_k \notin O_{sync}$), générer une macro de même nom que o_k, et générer les arguments de telle sorte que
      - ⋆ les noms d'arguments correspondent aux noms des sommets `alloc` a_m qui sont prédecesseurs ou successeurs de o_k ($\forall a_m / a_m \in \Gamma^{-1}(o_k) \quad ou \quad a_m \in \Gamma(o_k)$),
      - ⋆ l'ordre des arguments correspond à la valeur indexée par $Pos_{in}a_m$ (si a_m est prédécesseur) ou $Pos_{out}a_m$ (si a_m est successeur) dans la table associée à l'opération o_k.
    - ◇ si c'est une opération de synchhronisation ($o_k \in O_{sync}$), générer une macro de même nom que o_k, dont l'argument est le nom du sommet `alloc` prédécesseur si c'est une opération de type `Suc` ou du sommet `alloc` successeur si c'est une opération de type `Pre` .
  - générer une macro `endloop _`
  - pour chaque opération o_k ordonnancée sur p_l, jusqu'à la derniere opération :
    - ◇ générer une macro de même nom que o_k, sans générer d'arguments puisque ce sont les opérations d'entrée-sortie de finalisation,
- générer la macro `endmain _`

Exemple 7.3.1 *La figure 10.5 de la page 204 correspond au macro-exécutif généré selon ces règles pour une application de traitement du signal basée sur deux processeurs* `root` *et* `p` *dont les graphes d'algorithme, d'architecture et d'implantation sont donnés en exemple page 199.*

Chapitre 8

Transformation de macro-exécutif générique en exécutif

Sommaire

Dans le chapitre précédent nous avons construit les fichiers de macro-exécutif générique de chaque processeur à partir du graphe d'exécution. Il s'agit maintenant d'examiner le processus qui conduit à l'objectif final de la méthodologie AAA : l'exécution de l'algorithme sur l'architecture réelle. Cela inclu la transformation de chaque macro-exécutif en un exécutif compilable (donc écrit dans le langage du compilateur de chaque processeur cible de l'architecture), la compilation de cet exécutif et son chargement dans chaque processeur.

Nous avons vu que la transformation de chaque macro-exécutif en exécutif compilable repose sur des bibliothèques (Cf. § 5.2) d'exécutif qui composent le *noyau d'ex´ecutif de l'application.* Ces bibliothèques renferment les définitions de toutes les macro générées dans les fichiers d'éxecutif que nous venons de construire, certaine de ces définitions ont d'ailleurs été données en exemple (Cf. §7.2.1, § 7.2.1.1,§ 7.2.2.1, § 7.2.3.2).

8.1 Organisation du noyau d'exécutif

C'est pour faciliter le portage d'un noyau d'exécutif sur différentes architectures et réutiliser le plus de définition possible que nous avons modularisé ce noyau en plusieurs noyaux (Cf. 8.1). Ainsi, le noyau d'exécutif d'une application se décompose en deux grandes familles, les noyaux génériques et les noyaux non génériques :

- les noyaux génériques sont indépendants de l'application, ils renferment les définitions de toutes les opérations systèmes (allocations, synchronisations, communications etc) ajoutées au graphe d'algorithme

initial.

- les noyaux non génériques contiennent les définitions des opérations spécifiques à l'application.

Parmi les noyaux génériques, on distingue le noyau générique indépendant de l'architecture sous-jacente, et les noyaux d'exécutif spécifiques à chaque type de processeurs et à chaque type de communications qui composent l'architecture.

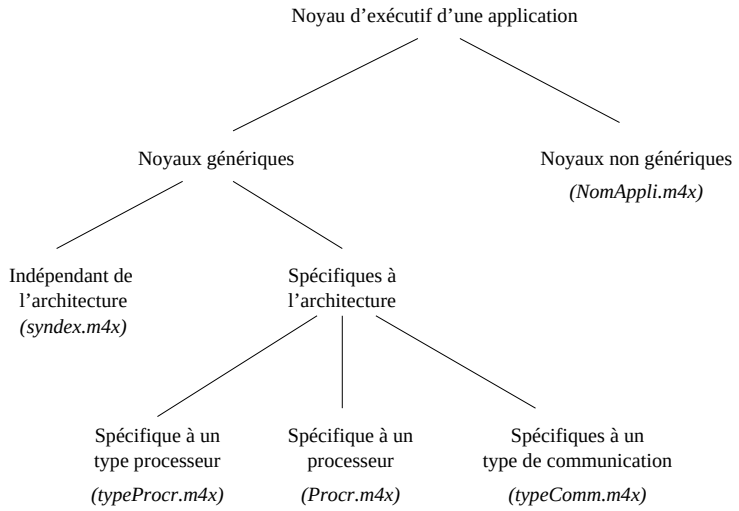


FIG. 8.1: *Noyaux d'exécutifs*

8.1.1 Noyau générique indépendant de l'architecture et de l'application

Ce noyau, nommé `syndex.m4x`, contient les définitions qui permettent de faire l'interface entre les macros génériques et les macros des noyaux spécifiques à l'architecture où à l'application. Il contient par exemple la définition (générique et indépendante de toute architecture) de `alloc` qui, commme nous l'avons vu selectionne la macro `BasicAlloc` spécifique à un type de processeur. C'est aussi ce noyau qui contient les définitions permettant de faire les vérifications de typage étudiées dans le chapitre précédent, ainsi que la gestion de messages d'erreurs.

8.1.2 Noyau générique spécifique à un type de processeur

Le noyau d'exécutif d'un type de processeur de l'architecture est systématiquement stocké dans un fichier de nom "typeProcr.m4x". Ce fichier doit contenir les définitions, dans le langage cible du compilateur qui sera utilisé pour ce type de processeur, des macros systèmes `main_`, `endmain_`, `loop_`, `endloop_`, `if_`, `else_`, `endif_`, `thread_`, `endthread_`, `spawn_thread_semaphore_`, `alloc_`, `Suc0_`, `Pre0_`. Ce noyau peut aussi contenir les définitions d'opérations de base standards (addition, filtre, etc).

Dans certains cas, il est possible que plusieurs noyaux de type de processeurs différents renferment les mêmes définitions pour certaines macros. Si nous prenons le cas de processeurs dont les compilateurs prennent un code source en langage C en entrée, de nombreuses définitions seront identiques pour chacun d'eux : l'allocation mémoire, les structures de contrôle etc. Seules les macros de synchronisations et de gestions de la mémoire partagée seront spécifiques aux processeurs (leur compilateur). Dans ce cas, plutôt

que de dupliquer inutilement une partie du contenu des noyaux, nous pouvons placer les définitions communes aux differents noyaux dans un autre fichier. C'est ainsi que nous avons construit un noyau `C.m4x` qui contient toutes les définitions standards du langage C. Il suffit ensuite, dans chaque noyau de type de processeur, d'inclure ce noyau commun par la directive m4 "include(C.m4x)".

En annexe : à titre d'exemple voici le contenu d'un noyau d'exécutif pour le langage C :

8.1.3 Noyau spécifique à un processeur

Si pour une raison quelconque il n'est pas souhaitable d'utiliser les définitions génériques d'un type de processeur, il est possible de définir des macros spécifiques à un processeur donné dans un noyau le "nomProcr.m4x". Ce noyau est optionnel dans la mesure ou les définitions nécessaires sont normalement déjà effectuées dans le noyau spécifique au type de processeur.

8.1.4 Noyau spécifique à un type de communicateur

Bien que les communicateurs fassent partie des processeurs, des communicateurs de processeurs différents peuvent avoir de nombreuses similitudes, notamment si ils sont connectés au même type de mémoire partagée. C'est pourquoi nous avons placé les macros d'exécutifs des communicateurs dans des fichiers différents de nom "typeComm.m4x". Le noyau d'exécutif d'un communicateur définit les macros de chargement arborescent des programmes `loadFrom _`, `loadOnto _`, les macros d'initialisation et de finalisation eventuelles **`type_comm_ini _`**, **`type_comm_end _`**, et bien sûr les macros de transfert de données `send _`, `recv _`, `sync _`.

Exemple 8.1.1 *Le fichier "TCP.m4x" donné en annexe correspond au communicateur permettant la communication interprocesseur par mémoire SAM que constitue le bus ethernet, émulé sous Unix par des threads (fonction* `fork()` *) de communication synchonisés avec le thread de calcul par l'intermédiaire d'une mémoire partagée.*

8.1.5 Noyau utilisateur

Ce noyau contient les définitions de toutes les macros des opérations de calcul ou d'entrée-sortie de l'algorithme qui ne font pas partie des opérations de bases standard des noyaux du processeur (typeProcr.m4x, nomProc.m4x), ce noyau est donc optionnel si les opérations de l'algorithme font toutes parties du noyau du type de processeur. Le nom de ce noyau est basé sur le nom de l'application : nomAppli.m4x. Les définitions de ces opérations peuvent être soit directement du code inséré en ligne, soit du code d'appel d'une fonction compilée séparement.

8.2 Automatisation des substitutions

Pour transformer le macro-exécutif d'un processeur en exécutif compilable dans le langage du compilateur de ce processeur, nous utilisons le macro-processeur m4 que nous avons présenté dans le chapitre précédent. La façon "standard" d'utiliser ce macro-processeur consiste à lui fournir, sur la ligne de commande, l'ensemble des bibliothèques contenant des définitions, puis à lui fournir le fichier contenant les macros à substituer. Prenons l'exemple d'une application "appli1" composée des processeurs $root$ et $p1$ dont les compilateurs acceptent le langage C en entrée et dont les communicateurs communiquent par bus CAN (donc de type CAN). Selon les indications que nous avons donné plus haut, la génération d'exécutif repose sur l'existence des bibliothèques C.m4x pour les processeurs, CAN.m4x pour les communicateurs, appli1.m4x pour les définitions des opérations spécifiques à l'algorithme de cette application. L'exécutif

du processeur root (fichier root.c) est obtenu (Cf. figure 8.2 à partir du macro-exécutif de ce processeur (root.m4) et de ces bibliothèques par la commande m4 suivante : `m4 C.m4x CAN.m4x appli1.m4x root.m4 >root.c` .

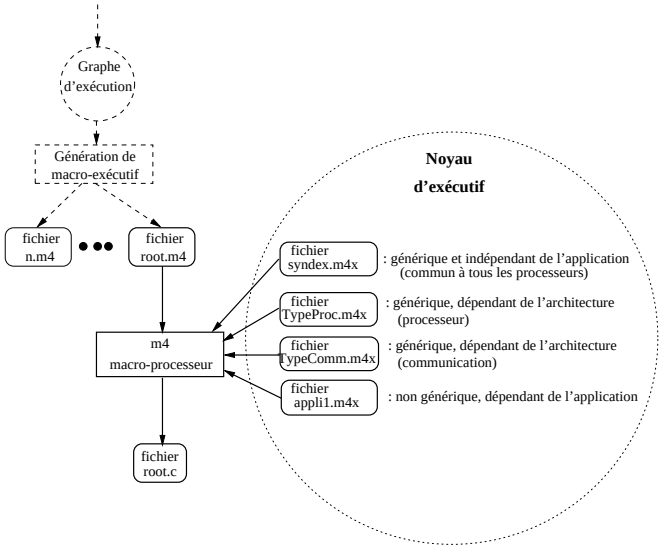


FIG. 8.2: *Génération d'exécutif*

Pour simplifier la tâche de l'utilisateur nous pouvons automatiser la génération de l'exécutif. Pour cela nous allons construire systématiquement une macro qui va utiliser les informations de types de processeurs et de noms d'applications qui font partie des arguments de la macro `processor_` (Cf. §7.2 et § 7.3) générée au début du fichier de macro-exécutif de chaque processeur, pour générer l'inclusion (à l'aide de la directive m4 `include(nomFichier)`) des fichiers correspondants. La définition de cette macro, utilisée pour la génération de tous les exécutifs, fait partie d'un fichier nommé "syndex.m4x" chargé automatiquement grâce à la directive m4 `include(syndex.m4x)` que nous avons inséré systématiquement (Cf. 7.2 et § 7.3) au début de chaque macro-exécutif (Cf. 7.2 et § 7.3).

La macro `processor_` du fichier syndex.m4x commence donc par réaliser ces fonctions de chargements :

```
# processor_(1typeProc,2nomProcr, 3anomAppli)
define('processor_', 'dnl
define('processorType_', $1)dnl peut servir plus tard pour d'autre macros,
define('processorName_', $2)dnl peut servir plus tard pour d'autre macros,
include($1.m4x) dnl macros du type de processeur $1
sinclude($3.m4x) dnl macros spécifiques à l application, optionnel
sinclude($2.m4x) dnl macros spécifiques au processeur, optionnel
')
```

Remarque 49 `sinclude(nomFichier)` *est une directive de m4 qui comme* `include(nomFichier)` *permet d'inclure des définitions faisant partie d'un fichier "nomFichier", mais dans le cas de* `sinclude` *, il n'y a pas de génération de message d'erreur si le fichier n'existe pas, la directive est simplement ignorée. Nous l'utilisons donc pour charger les bibliothèques optionnelles que sont le noyau utilisateur et le noyau spécifique à un processeur.*

Le chargement des noyaux d'exécutifs spécifiques aux types de communicateurs est réalisé par la macro thread_(type_comm, nom_comm, <ListeNomsProcesseurSconnectésComm>) (Cf. § 7.2.3) qui précède chaque séquence de communication de chaque communicateur, puisque cette macro reçoit comme argument de type de communicateur. Seul le noyau "syndex.m4x" (donné en annexe) est indépendant de toute cible. Il contient d'autre définitions qui permettent la génération automatique de commentaires, d'erreurs ainsi que la mise en place de vérifications d'existence et de type de données. Ces macros ne seront pas abordées ici car elles permettent surtout de faciliter la mise au point des exécutifs. Cependant, il est important de noter que pour permettre ces vérifications, il faut enrichir la plupart des macros présentées dans cette partie, c'est pourquoi les exécutifs présentés en annexe sont relativement plus complexes que les exemples présentés ici.

8.3 Chaîne de compilation : génération de makefile

Après avoir généré le code source dans le langage de chaque processeur à l'aide des noyaux d'exécutifs correspondant, on utilise la chaîne classique de compilation. En mono-processeur on commence par la compilation du code source (C, Fortran, assembleur etc) en un code en langage assembleur du processeur. Ce code est à son tour transformé en code "objet" binaire relogeable qu'il faut ensuite "linker" (éditer des liens) pour obtenir un code exécutable. L'édition de lien permet d'assembler des morceaux de codes-objets compilés séparement (pour l'utilisation de fonctions déjà programmées par exemple). Le binaire executable obtenu n'a plus qu'à être chargé dans la mémoire du processeur cible grace à un *loader*. Si la mémoire du système est programmable sur place (in situ), le loader est un simple logiciel qui se contente de lire un fichier pour programmer une mémoire (souvent effaçable électriquement). Sur d'autres architectures, la mémoire du processeur ne peut être programmée sur place et doit être programmée ailleurs à l'aide d'outils matériels et logiciels séparés. Le loader peut ainsi correspondre à un programmateur d'EPROM, il nécessite une intervention humaine à chaque modification du contenu du programme ce qui est peu confortable dans le cadre du prototypage rapide. Quand l'architecture est distribuée et hétérogène, il faut autant de compilateurs qu'il y a de types différents de processeurs, de plus le chargement des programmes nécessite un loader distribué hétérogène plus complexe. Ce loader distribué fait partie de l'exécutif que nous générons pour chaque processeur, il correspond aux macros loadFrom_(nomProcr, listeCommunicateur) et loadOnto_(nomComm, listeCommunicateur) de chargement arborescent des programmes que nous avons étudié dans la section 7.2.3.4, p. 165.

Pour accélérer le processus de compilation, et minimiser les erreurs de manipulation, il est toujours préférable d'utiliser un processus de compilation contrôlé par un *makefile*. Un makefile est un script dédié à la gestion de la chaîne de compilation, il repose sur l'outil gmake [78], [73]. Dans le cas d'une application composée de plusieurs types de processeurs différents, l'écriture du makefile peut s'avérer très complexe puisque faisant intervenir différents types de compilateurs avec des options différentes et un processus de chargement et d'exécution du code spécifique aux interconnexions entre les processeurs de l'architecture. Nous proposons donc de générer automatiquement le makefile correspondant à l'application. Pour cela, il faut fournir un certain nombre d'informations relatives à la chaîne de compilation de chaque processeur de l'architecture, mais aussi relatives aux connexions entre le processeurs pour construire et charger les exécutables de chaque processeur dans un ordre cohérent avec celui des macros loadFrom_(nomProcr, listeCommunicateur) et loadOnto_(nomComm, listeCommunicateur) faisant parties des séquences de communication de chaque processeur.

Les informations spécifiques à la chaîne de compilation (nom et emplacement du compilateur, option de compilation, nom du linker, etc) de chaque processeur de l'architecture sont spécifiées dans un fichier dont le nom est celui du type de processeur et dont l'extension est ".m4m" (pour m4 Makefile).

Les informations relatives aux interconnexions entre les processeurs du graphe de l'architecture sont

générées automatiquement dans un fichier qui porte le nom de l'application et dont l'extension est ".m4".

Ensuite, à partir des informations contenues dans ces deux fichiers, il est possible à l'aide du macro-processeur `gm4` et d'une bibliothèque indépendante de toute architecture que nous avons écrite (`syndex.m4m`), de générer automatiquement le makefile spécifique à l'application. La figure suivante (Cf. 8.3) illustre le mécanisme qui permet d'automatiser la chaîne de compilation.

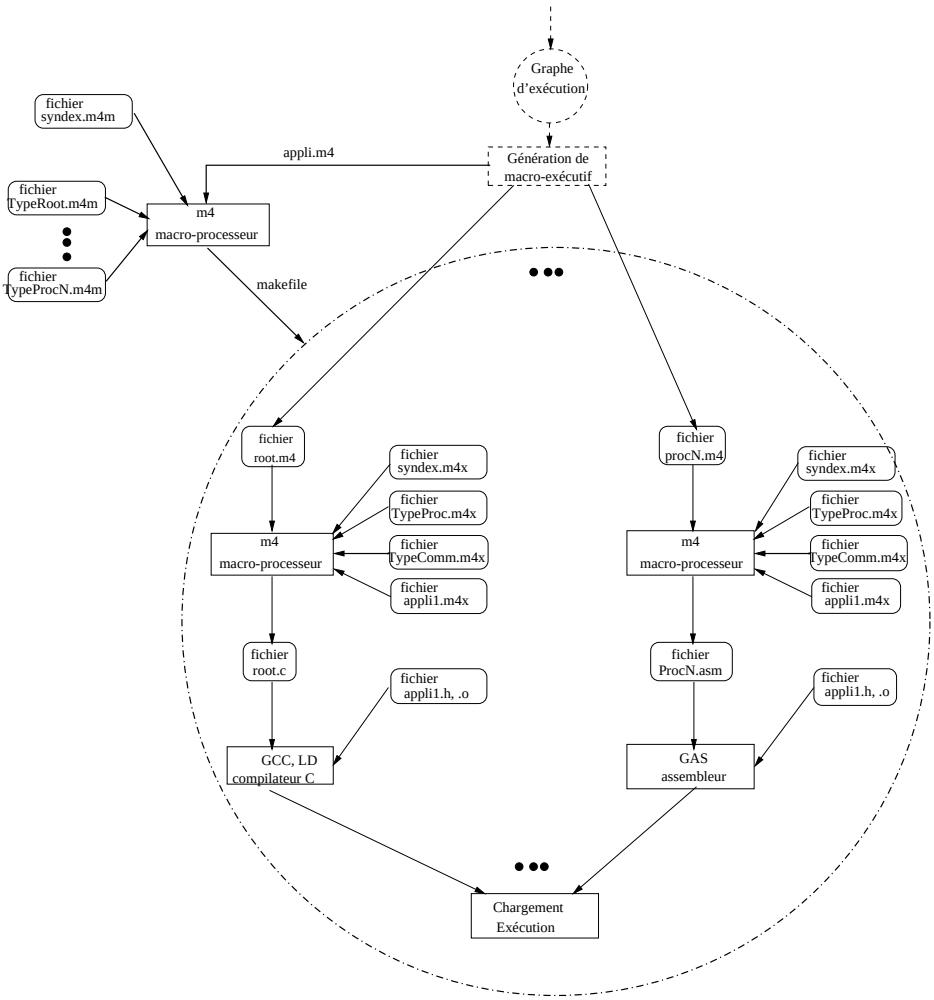


FIG. 8.3: *Automatisation de la chaîne de compilation*

Troisième partie

Développement logiciel

Chapitre 9

Etat de l'art des outils existants

Dans ce chapitre, nous avons essayé de sélectionner les outils les plus proches de nos objectifs : le prototypage rapide d'applications temps réel sur des architectures hétérogènes distribuées. Tous ces outils permettent d'aider à l'implantation d'algorithme sur des architectures parallèles homogènes ou hétérogènes.

9.1 CASCH

CASCH[59] (dont l'acronyme est **C**omputer **A**ided **SCH**eduling) est un outil graphique universitaire développé à l'université de Hong Kong par l'équipe du Professeur I. Ahmad. Il a pour but d'automatiser l'extraction de parallélisme d'algorithmes (traitement du signal, vision) spécifiés textuellement ou séquentiellement pour les exécuter sur des machines parallèles. Cet outil propose une grande bibliothèque extensible d'algorithmes de partitionnement et d'ordonnancement. L'utilisateur peut ainsi rechercher parmi les algorithmes implantés, celui qui génère l'ordonnancement qui utilise au mieux les ressources. Chaque algorithme peut être interactivement analysé, testé et comparé en utilisant des données fournies par un simulateur. Pour faciliter ce travail de comparaison d'algorithmes de partitionnement et d'ordonnancement, CASCH possède aussi un générateur automatique et aléatoire de graphes.

9.1.1 Algorithme

L'algorithme à implanter est d'abord spécifié sous forme textuelle séquentielle (type Fortran). C'est une séquence de boucles imbriquées avec appel de procédures. Cette description est automatiquement transformée en un graphe acyclique orienté (DAG) par un *parser* et un *lexer* incorporés au logiciel. CASCH permet aussi de spécifier directement des DAGs au moyen de son interface graphique. CASCH n'a pas de notion de retard, ni de notion de conditionnement. Le parallélisme est extrait de la spécification textuelle séquentielle par transformation de cette spécification en un graphe qui met en évidence le parallélisme de l'algorithme. Cette transformation déroule complètement les boucles, chaque noeud du graphe obtenu correspond à un appel de procédure, les arcs représentent les dépendances de données entre appels de procédures. La taille du grain est fixe et correspond à l'appel de procédure à l'intérieur de boucles. Il n'est pas possible de décrire un noeud par un sous graphe.

9.1.2 Architecture

CASCH gere uniquement les architectures homogènes, elles sont spécifiées par des graphes. La topologie d'interconnexion est libre, mais n'est pas utilisée pour les optimisations temporelles. Elle repose sur

des communications synchrones point-à-point. Il n'y a pas de support pour la description de hiérarchies de mémoires. Les travaux publiés ont été expérimentés sur une architecture cible de type Intel Paragon.

9.1.3 Adéquation

Dans CASCH, le DAG généré est partitionné automatiquement par un algorithme choisi par l'utilisateur parmi trois classes (UNC *unbounded number of clusters*, BNP *bounded number of processors*, APN *arbitrary processor network*). Ces algorithmes prennent en compte les durées d'exécution des procédures, les durées des communications (minimisent les communications interprocesseurs par regroupement de procédures).

Ainsi, chaque noeud du graphe de l'algorithme possède un poids qui correspond à la durée d'exécution de la procédure qu'il représente. Chaque arc du graphe possède aussi un poids qui correspond à la durée de transmission du message qu'il représente. Quand les deux procédures connectées à chaque extrémité d'un arc sont ordonnancées sur le même processeur, le poids de cet arc devient nul. L'estimation des durées de communication (obtenue expérimentalement) est basée sur le coût de chaque primitive de communication (*send, receive, broadcast*). Elle est estimée en utilisant le temps de *startup*, la longueur du message et la bande passante du canal de communication. Les poids des noeuds et des arcs sont obtenus par un module de CASCH (*estimator*) qui est spécifique à chaque architecture cible.

L'approche utilisée pour équilibrer la charge des processeurs et minimiser les communications consiste à partitionner le graphe de l'algorithme afin de regrouper les procédures en autant de tâches qu'il y a de processeurs. Chaque tâche est ensuite assignée à un processeur. L'outil effectue ainsi un ordonnancement statique des procédures, et des communications. La capacité mémoire est Post-évaluée. Les synchronisations entre les tâches exécutées sur différents processeurs sont assurées par des primitives de communication. Les primitives de base pour échanger des données entre processeurs sont *send* et *receive*. Elles peuvent être insérées automatiquement selon la procédure suivante : après partitionnement et distribution chaque noeud du macro graphe flot de données a été alloué à un PE (Processeur Élémentaire). Si un arc quitte un noeud A pour un noeud B qui appartient à un autre PE, la primitive *send* est insérée après le noeud A. De la même façon, si un arc arrive sur un noeud B en provenant d'un noeud A appartenant à un autre PE, la primitive *receive* est insérée avant le noeud B. Si le message a déjà été envoyé à un PE donné il n'est pas nécessaire de le renvoyer à ce PE. Le *broadcasting* ou le *multicasting* peuvent être utilisés plutôt que des messages multiples. Cette méthode n'assure pas que la séquence de communication soit correcte, il faut donc utiliser une stratégie dite de *send first* pour ré-ordonner les primitives de communication, en l'occurence les *receive* sont ré-ordonnés selon l'ordre des *sends*. Les communications à destination d'un même processeur peuvent être groupées (*"packed"*) pour gagner le temps de startup.

CASCH permet pour chaque processeur, de prédire et d'afficher graphiquement l'utilisation du processeur. Il fournit aussi le temps passé en calcul et le temps passé en communication de chaque processeur.

Il est important de noter que le routage des communications est effectué dynamiquement par le système d'exploitation sous-jacent.

9.1.4 Génération d'exécutif

CASCH génère automatiquement du code Fortran à partir des résultats des algorithmes de partitionnement-ordonnancement. Ce code est constitué d'appels de procédures utilisées dans la spécification initiale, il consiste en un déroulement des boucles. Ce code repose sur un système d'exploitation qui doit gèrer le routage dynamique des communications, il ne génère donc pas d'exécutifs. Aucun autre détail n'est donné sur le système d'exploitation. Les exemples présentés dans les articles étudiés ne montrent pas de génération automatique de boucles, mais uniquement des appels de procédures avec insertion de primitives de communication.

9.1.5 Conclusion

Cet outil ne cible pas les domaines de l'embarqué et du temps réel. C'est un outil d'aide à l'équilibrage de charge statique de réseau de station ou de machines multiprocesseurs homogènes dans le contexte d'applications de traitement du signal et des images. C'est essentiellement un outil qui permet l'évaluation et la comparaison d'algorithmes de partitionnement et d'ordonnancement de graphes flots de données.

9.2 GEDAE

GEDAE (Graphical Entry, Distributed Applications Environment [62]) est un logiciel commercial graphique d'aide au développement d'applications de traitement du signal (systèmes temps réel) sur architectures multiprocesseurs embarquées. Le developpement initial de GEDAE a commencé grâce à un financement fournit par le gouvernement américain dans le cadre du projet RASSP [85] (Rapid Prototyping of Application Specific Signal Processors). GEDAE a été utilisé pour supporter le développement des applications radar et sonar. Suite à son utilisation dans le programme RASSP, Lockheed Martin ATL a décidé de commercialiser GEDAE afin d'assurer sa pérennité et son évolution. GEDAE couvre tout le domaine du Traitement de Signal (TS) sans restriction apparente et permet de spécifier hiérarchiquement les applications à partir de bibliothèques de fonctions usuelles, de développer des fonctions utilisateur, de placer les traitements sur des architecture parallèles, de décrire les transferts de données en tenant compte du parallélisme des traitements, de générer automatiquement du code C, de simuler l'execution sur station de travail ainsi que le contrôle temps réel de l'exécution (visualiser la place mémoire occupée par chaque traitement, visualiser le temps passé à exécuter chaque traitement). GEDAE est développé pour les PC sous Windows NT 4.0 et pour les stations de travail Sun sous Solaris 2.5.1 avec X11R5, Motif 1.2. Que lque soit le système hôte utilisé, un compilateur C ANSI est nécessaire pour chaque type de processeur cible de l'architecture utilisé.

9.2.1 Algorithme

GEDAE repose sur un formalisme de graphe flot de données, conditionnel, acyclique. La notion de retard est modélisable à l'aide des primitives cycliques. GEDAE fournit un environnement hiérarchique pour spécifier les algorithmes. L'approche est aussi bien top-down que bottom-up. On décompose l'application en boîtes fonctionnelles réutilisables. Ces boîtes donnent une structure modulaire aux graphes de flots ; elles peuvent comprendre d'autres boîtes hiérarchiques, des boîtes primitives, des entrées, des sorties, des paramètres et des équations mais l'imbrication ne peut être récursive. Les boîtes primitives sont "atomiques" d'un point de vue placement ; c'est-à-dire qu'on ne peut les distribuer que sur un unique processeur (objets de la granularité la plus fine).

9.2.2 Architecture

L'architecture est décrite par un unique fichier de configuration, elle contient 4 sections :

1. la section processeur (processeurs composant le système embarqué) est composée du nom logique (entier) de chaque processeur, du nom physique (chaîne de caractère) de chaque processeur, du nom du type de processeur,

2. la section communication (mécanismes de communication inter-processeurs) est composée de la description des types de communications (socket, mémoire partagée, passage de messages, accès direct à la mémoire), de la liste de processeurs pour ce type (en extension ou en intension, connexion unidirectionnelle ou bidirectionnelle), de la mémoire,

3. la section mémoire décrit chaque type de mémoire employé (Distributed RAM, Shared RAM) et le nombre d'emplacements mémoire accessibles (en octets) pour chaque type de mémoire,

4. la section "système" permet d'insérer des paramètres d'initialisation (chaîne de caractères) spécifiques au matériel.

9.2.3 Adéquation

GEDAE ne fournit pas d'outils automatiques d'adéquation entre l'algorithme et l'architecture. Le partitionnement et la distribution sont donc donnés graphiquement par l'utilisateur. Les applications GEDAE peuvent être distribuées sur un réseau de stations de travail, sur un système de processeurs embarqués ou sur une station de travail multi-processeurs. La technique de distribution est identique pour les deux dernières. La distribution d'une application sur un réseau de stations de travail requiert que l'application soit conçue à l'aide de boîtes fonctionnelles primitives hôtes. Dans le cas embarqué, il faut diviser l'application en "groupes embarquables". Un groupe embarquable est un sous-graphe du graphe principal qui est un ensemble de boîtes embarquables connectées. Un groupe ainsi défini réagit comme si c'était une boîte fonctionnelle unique. C'est le plus large segment d'un graphe à l'intérieur duquel on dispose d'un contrôle complet sur les caractéristiques d'exécution, soit le plus grand ensemble de boîtes embarquables qui sont connectées par le flot de données. Plus précisément, les groupes sont des ensembles de boîtes fonctionnelles primitives embarquables connectées qui ont pour bornes des boîtes hôtes. Les boîtes hôtes s'exécutent toujours sur le processus hôte. Ce dernier envoie ou reçoit des données aux processus embarqués via des queues insérées par GEDAE. Les groupes peuvent être aussi vus comme des collections de fonctions connectées ordonnançables statiquement qui peuvent toutes être placées dans un même ordonnancement. Les objets qui peuvent seulement être ordonnancés dynamiquement sont des délimiteurs de groupes.

GEDAE ordonnance les fonctions aussi bien statiquement (librairie embarquable) que dynamiquement (librairies embarquable et hôte). L'ordonnancement est construit en privilégiant l'aspect statique tout en conservant l'aspect dynamique (i.e. déterminé à l'exécution) lorsque cela s'avère nécessaire. La séquence d'exécution des fonctions est déterminée par un ordonnanceur local.

Dans le cas d'une distribution sur un réseau de stations de travail, l'ordonnancement de l'exécution des fonctions est traité automatiquement par GEDAE. En revanche, l'exécution embarquée nécessite la création d'un ordonnancement d'événements à exécuter sur chaque processeur. Cet ordonnancement est utilisé pour générer automatiquement le code du programme en C puis les exécutables chargés sur les processeurs embarqués. Une application peut avoir de multiples ordonnancements. Chaque ordonnancement statique est distribué sur un processeur dans une mémoire différente.

9.2.4 Génération d'exécutif

GEDAE génére un exécutif codé en C ANSI pour chacun des processeurs embarqués à partir du partitionnement et de la distribution utilisateur. Un noyau temps réel GEDAE porté sur la machine cible et résident sur chacun des processeurs embarqués exécute l'application auto-codée.

La première étape du processus de génération automatique d'exécutif dans le cas embarqué est de créer un ordonnancement dynamique pour le groupe dans sa globalité en ignorant initialement les partitions. Des ordonnancements statiques, qui communiquent par des queues dynamiques, sont générés pour chaque sous-groupe. L'exécution de l'ordonnancement statique de chaque sous-groupe est dynamiquement ordonnancé selon la disponibilité de données dans les queues entre sous-groupes.

La seconde étape consiste à partitionner le groupe. On subdivise l'ordonnancement dynamique en un ordonnancement dynamique par partition. Cela implique d'allouer des queues particulières à ces ordonnancements et de subdiviser les ordonnancements statiques et les sous-ordonnancements de chaque sous-groupe.

Les communications sont générées et traitées automatiquement par GEDAE qui supporte les communications point-à-point et bus via des fonctions “send-receive”. Dans le cas d’une distribution sur un réseau de stations de travail, tous les transferts de données entre stations de travail sont effectués via le réseau par sockets et threads ; tous les transferts de données entre fonctions sur une station de travail sont réalisés par copies mémoire. Dans le cas embarqué, il y a des transferts de données entre partitions du même groupe et entre divers points d’un groupe vers un autre. Le mécanisme de communication dépendant de l’architecture utilisée par chacun de ces transferts de données inter-processeurs peut être sélectionné à partir des différents mécanismes supportés par le matériel cible.

Pour le moment, GEDAE génère des exécutifs uniquement pour les cartes COTS Mercury, Alex et Ixthos. L’utilisation d’autres cartes COTS ou de cartes dédiées nécessite une prestation spécifique de Lockheed Martin ATL. L’outil n’est donc pas “ouvert” au niveau de la description de l’architecture.

La simultation est un des points forts de GEDAE, il existe des outils de visualisation du signal comme l’oscilloscope, l’analyseur spectral, etc. ; il suffit de se connecter quelque part dans le flot de données et une fenêtre affiche la forme du signal. On peut observer l’activité du graphe, du coté matériel ou logiciel, de manière intrusive ou non : dans le premier cas, le rafraichissement de l’affichage se fait en permanence, ce qui ralentit l’exécution alors que dans le second, l’information est stockée en tâche de fond dans un buffer circulaire et la trace est affichée à la demande de l’utilisateur pour une analyse ultérieure. Si le graphe a été exécuté, l’information concernant le temps est mesurée, sinon elle est estimée.

9.2.5 Conclusion

GEDAE est connecté aux outils de l’environnement de co-design de Lockheed Martin ATL : COSMOS pour le choix d’architecture à partir de modèles de performance en VHDL, ObjectGEODE pour le développement de la partie contrôle de l’application TS. C’est un des outils les plus complets et les plus aboutis que nous ayons trouvé dans ce domaine. Cependant, si son aspect graphique permet de construire aisément des algorithmes, de les implanter et d’étudier leur évolution, l’ensemble des architectures cibles pour cette implantation n’est pas extensible par l’utilisateur. De plus l’outil n’effectue aucune optimisation automatique, ni au niveau de la distribution-ordonnancement de calculs, ni au niveau de l’optimisation de la mémoire.

9.3 Ptolemy II

PtolemyII [100, 24] est le successeur de Ptolemy 0.7 (appelé aussi “Ptolemy Classic”[8, 15])dont il reprend de nombreuses caractéristiques. PtolemyII est un logiciel de recherche, non commercialisé, développé par le ”Department of Electrical Engineering and Computer Sciences” de l’université de Californie Berkeley, sous la direction de Edward A. Lee. PtolemyII vise la conception et la simulation, hétérogènes et concurrentes, de systèmes embarqués réactifs. Son but est de supporter la construction et l’interopérabilité de modèles exécutables selon des modèles d’exécution (“domaines”) très variés. PtolemyII repose extensivement sur la technologie objet Java, aussi bien pour l’environnement de développement et son interface graphique ”Diva”, que pour le codage (et donc l’interopérabilité) des modèles de simulation, jusqu’à l’implantation sur un système réel (applets ou Java embarqué).

PtolemyII vise le domaine des systèmes embarqués complexes, qui combinent plusieurs technologies, comme par exemple de l’électronique analogique et numérique, du matériel et du logiciel, des composants électroniques et mécaniques, et qui combinent également plusieurs types de traitements, comme le traitement du signal, les asservissements en boucle fermée, les automates de décision séquentielle, ou les interfaces utilisateur. L’accent est mis surtout sur la simulation de ces systèmes, selon plusieurs niveaux de raffinement successifs; le codage d’une implantation exécutable en temps réel par le système cible embarqué n’est vu

que comme le dernier niveau de raffinement, qui nécessite, comme les autres, le support d'une machine virtuelle Java.

La construction des modèles est spécifiée au moyen de graphes hiérarchiques, vus comme une syntaxe abstraite commune à tous les blocs-modèles, dont la sémantique dépend du modèle d'exécution attaché à chaque niveau hiérarchique.

L'interface et l'implantation d'un bloc-modèle sont séparées afin que ses différentes implantations (à des niveaux de simulation de plus en plus fin, jusqu'au code embarqué) aient la même interface pour être interchangeables.

PtolemyII permet une modification cohérente des modèles en cours de simulation.

Pour déterminer les types des ports d'interconnexion, un raffinement monotone sur un ordre partiel entre types est utilisé.

9.3.1 Algorithme

Le credo de PtolemyII est que la variété des domaines d'application visés nécessite une variété de formalismes de spécification, chacun bien adapté à un domaine spécifique. La décomposition d'un système complexe en sous-systèmes est soumise à des règles d'interaction entre sous-systèmes, qui définissent la sémantique du "modèle d'exécution" ou "domaine" du système. La décomposition récursive d'un système peut être hétérogène, c'est-à-dire que chaque décomposition peut avoir un modèle d'exécution différent. La décomposition récursive d'un système est décrite par un graphe hiérarchisé ("clustered hierarchical graph"), dont chaque sommet, appelé "entitée", est représenté graphiquement par une boîte comportant des "ports" d'interface entre l'extérieur et l'intérieur de la boîte (qui peut contenir un sous-graphe), et dont chaque arête, appelée "relation", est représentée graphiquement par une interconnexion en étoile entre ports.

PtolemyII définit les "domaines" suivants :

- **CT** ("Continuous Time") Chaque sommet représente un sous-système continu modélisé par un système d'équations différentielles ($dx/dt = f(x,u,t)$ et $y = g(x,u,t)$ où x est l'état du sous-système, u son entrée, y sa sortie, t le temps continu, et dx/dt la dérivée de l'état par rapport au temps), et chaque arc représente une variable continue. Ce domaine est simulé par différentes méthodes de résolution numérique des équations différentielles (Euler, Runge-Kutta, trapèzes), avec un support pour l'interaction avec les domaines discrets (DE, SC),

- **PN** ("Process Network") Chaque sommet représente un processus séquentiel, et chaque arc représente un canal de communication monodirectionnel FIFO "asynchrone", c'est-à-dire où les données sont reçues dans l'ordre où elles ont été émises, avec blocage du processus en réception si FIFO vide (modèle de Kahn). De plus, le temps est global et s'écoule pour un processus soit lorsqu'il est bloqué en attente d'une réception, soit lorsqu'il attend explicitement que le temps s'écoule. L'implantation de ce domaine est basé sur les threads Java.

- **DFM** ("Design Flow Management") Chaque sommet représente un outil de conception, et chaque arc représente une interface entre outils (fichier, pipe, CORBA); DFM est une utilisation spécifique de PN pour automatiser le flot de conception.

- **CSP** ("Communicating Sequential Processes") Restriction "synchrone" du domaine PN, avec rendez-vous entre le processus émetteur et le processus récepteur d'une même communication: le premier prêt doit attendre l'autre (modèle de Hoare).

- **DE** ("Discrete Event") Chaque sommet est un "acteur", qui reçoit et émet par ses arcs adjacents des "jetons" (événements) datés lors de leur émission. En réaction à la réception d'un jeton, un acteur

peut modifier son état interne et émettre à son tour d'autres jetons. Un ordonnanceur garantit un traitement chronologique des jetons.

- **SDF** ("Synchronous DataFlow") Restriction déterministe du domaine DE, où chaque acteur doit recevoir un nombre prédéterminé invariable de jetons sur chacun de ses arcs d'entrée avant de réagir, et produit un nombre prédéterminé invariable de jetons sur chacun de ses arcs de sortie à chaque réaction. Cette restriction permet de prédéterminer un ordonnancement simple moins coûteux qu'un ordonnancement multitâches.

- **OD** ("Ordered Dataflow") Relaxation par rapport au domaine DE de la notion de temps global : le temps est local à chaque acteur, qui doit en conséquence arbitrer l'ordre de traitement de jetons reçus portant la même date.

- **SC** ("Star Charts") Domaine des machines à états finis (FSM) hiérarchiques, où chaque sommet est un état exclusif des autres, qui peut être un sous-système se décomposant dans le domaine SC en FSM, ou dans un autre domaine (par exemple CT ou DE ou SDF), et où chaque arc est une transition d'état déclenchée par un événement externe, ou par un événement interne détecté dans le sous-système de l'état courant.

Cette liste n'est pas exhaustive et peut s'enrichir dans le futur.

9.3.2 Architecture

Elle est entièrement à la charge de l'utilisateur, qui doit la simuler soit implicitement soit explicitement dans ses modèles. Le modèle de programmation repose sur les thread Java de la machine virtuelle Java.

9.3.3 Adéquation

La distribution spatiale de l'algorithme n'est pas automatique, elle doit être spécifiée par l'utilisateur, PtolemyII n'automatise donc pas cette tâche. L'ordonnancement des calculs et des communications est effectué dynamiquement par l'ordonnanceur de chaque sous-domaine.

9.3.4 Génération d'exécutif

PtolemyII crée et interconnecte des objets Java, en interaction avec l'utilisateur pour construire le graphe hiérarchisé représentant son application. Ces objets sont les acteurs directs de la simulation, leur ordre relatif d'exécution est déterminé par l'ordonnanceur de chaque sous-domaine en fonction de leurs interconnexions. Le type et la sémantique des communications sont aussi fonction du domaine. Leur exécution est supportée par la machine virtuelle Java. Pour toute partie de l'application réelle qui ne serait pas supportée par une machine virtuelle Java, la traduction du code Java de simulation en code cible (assembleur, VHDL, plans mécaniques...), ainsi que la vérification de leur équivalence, sont à la charge de l'utilisateur.

Comme pour GEDAE, la simulation est un des points fort des versions de Ptolemy, il propose de nombreux outils de visualisation graphique des signaux (package PtPlot). Il permet la simulation dynamique, visant même l'édition interactive de l'application en cours de simulation.

9.3.5 Conclusion

PtolemyII s'intéresse surtout à la simulation d'applications complexes très hétérogènes, mais apparemment pas à l'optimisation ni à la génération automatiques de leur implantation.

9.4 TRAPPER

TRAPPER[91] est un outil commercial distribué par ENIAS Software. C'est l'acronyme de **TRA**ffonic **P**arallele **P**rogramming **E**nviRonment (TRAFFONIC est un programme de recherche Daimler-Benz sur l'utilisation de l'électronique dans les véhicules). TRAPPER est un environnement de développement graphique pour les systèmes embarqués de type MIMD, issu d'une collaboration entre le laboratoire de recherche Daimler-Benz et le Centre Allemand National de Recherche (GMD). Il est basé sur le modèle de programmation des processus séquentiels communicants (CSP) et permet la conception, le placement, la visualisation et l'optimisation de systèmes parallèles. Le principe retenu dans TRAPPER est de laisser au programmeur le soin de spécifier explicitement la structure parallèle de son programme sous forme d'un graphe de processus et de l'aider dans les différentes phases de développement du système embarqué.

TRAPPER supporte les architectures à base de Transputers ainsi que les architectures cibles à base de PowerPc (par le biais des outils Parsytec PowerTools[81]) ainsi que les réseaux de stations de travail par le biais des bibliothèques PVM[37] et MPI[92].

Le logiciel TRAPPER est constitué de 4 modules, chacun dédié à une étape du développement (Cf. figure 9.1) :

- *DesignTool* : pour spécifier l'algorithme à implanter,
- *Confi gTool* : pour configurer l'architecture et effectuer le "mapping" de l'algorithme sur l'architecture,
- *VisTool* : pour observer le comportement de l'algorithme afin de déboguer son implantation parallèle,
- *PerfTool* : pour observer le comportement de l'architecture lors de l'exécution de l'algorithme afin d'effectuer des optimisations.

9.4.1 Algorithme

Le module *DesignTool* permet de spécifier graphiquement l'algorithme sous la forme d'un graphe de processus qui exprime le parallélisme du programme. Chaque processus (noeud du graphe) est une tâche séquentielle décrite par une représentation textuelle (langage C) qui a uniquement accès à sa mémoire locale. Ces processus communiquent par passage de messages au moyen de canaux de communication (arcs du graphe).

Le graphe de processus peut être décrit de façon hiérarchique : chaque noeud ou bloc peut lui même représenter un sous-graphe de processus dont chaque élément peut être fait à son tour d'autres graphes. Chaque sous-système peut être ainsi décomposé jusqu'à atteindre le niveau d'une description séquentielle (code exécutable).

Un processus est caractérisé par des propriétés (attributs) "héritées" ou communes avec d'autres processus ainsi que des propriétés propres : chaque processus possède un identificateur unique et un champ pour le type du processus. C'est par ce dernier que l'on associe le code séquentiel à un processus (plusieurs processus peuvent donc être de même type et ainsi faire appel à un même code, TRAPPER supporte ainsi implicitement l'approche SPMD), d'autres propriétés sont stockées au moyen du champ "type de processus", par exemple la taille du tas pour ce code, la valeur d'attribut autorisant le "traçage" ou non etc.

Les attributs spécifiques à chaque instance permettent de particulariser les processus faisant partie du même type, parmi ceux-ci on trouve l'attribut *time* qui permet de spécifier une estimation de la charge processeur qu'occasionnera le processus et sera donc utilisé en phase d'adéquation, et l'attribut *NodeType* qui permet optionnellement de spécifier un processeur sur lequel on veut que soit exécuté le processus. Si l'attribut est vide, le placement du processus sera fait automatiquement par le module *Confi gTool.*

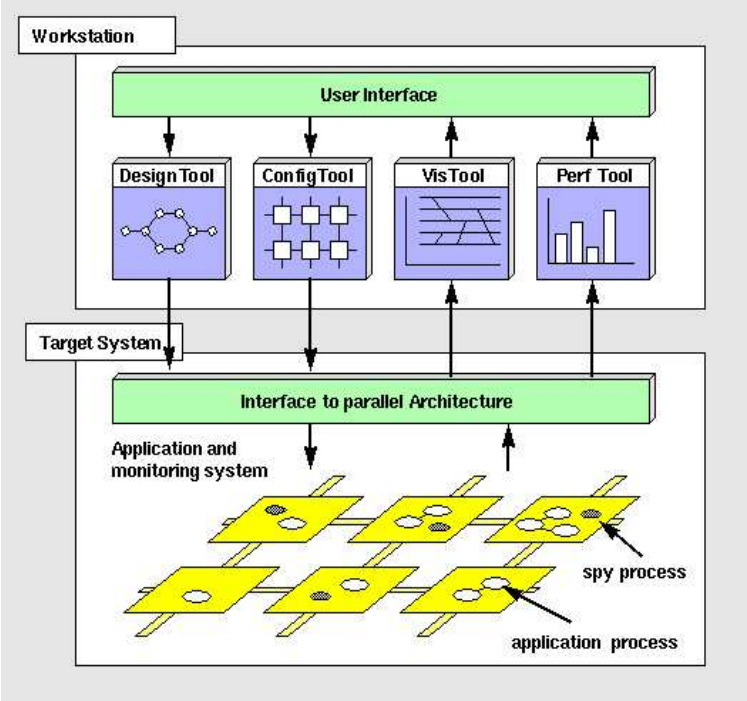


FIG. 9.1: *Structure et fonctionnalités de Trapper*

Les arcs du graphe de processus (symbolisant le passage de messages entre processus) peuvent avoir un attribut qui est la charge estimée de communication. Si l'attribut est renseigné par l'utilisateur, l'algorithme de placement automatique pourra équilibrer la charge de communication.

9.4.2 Architecture

Le module *ConfigTool* permet à l'utilisateur de spécifier graphiquement la configuration de l'architecture matérielle hétérogène cible sous la forme d'un graphe matériel (*Hardware Graph*) où les noeuds représentent les processeurs et les arcs connectant les noeuds représentent les canaux de communication. Ce graphe peut être hétérogène. Trapper supporte actuellement les processeurs basés sur la technologie des transputers (T2xx, T4xx, T8xx, C004), ainsi que les PowerPC (à travers les outils Parsytec[81]) et les réseaux de stations de travail (à travers PVM[37] et MPI[92]).

Les arcs et les noeuds peuvent être valués afin de permettre au module de placement (*ConfigTool*) d'équilibrer la charge de chaque processeur et de chaque lien de communication. Pour cela l'architecture doit être caractérisée en renseignant les différents attributs des processus (charge de calcul qu'il va entraîner pour un processeur, taille du tas demandée par le processus, etc) et la charge éventuelle que va entraîner chaque message sur un lien de communication. Ce sont des estimations que l'utilisateur pourra confronter à la réalité après récupération des traces d'exécution (fournies par les modules *VisuTool* et *PerfTool*).

9.4.3 Adéquation

Le module *ConfigTool* fournit deux modes pour l'allocation spatiale des processus sur les processeurs : manuel ou automatique. En mode manuel c'est l'utilisateur qui, pour chaque processus, spécifie le processeur qui l'exécutera. En mode automatique, TRAPPER utilise les informations de charges fournies lors

de la description de l'algorithme (poids donné aux noeuds et aux arcs) avec l'objectif double d'équilibrer la charge de chaque processeur et de minimiser les communications. La solution est déterminée par une heuristique qui n'est pas décrite. Le résultat est affiché graphiquement en coloriant de la même couleur les processus et les processeurs qui les exécutent. L'heuristique prend en compte la charge processeur qu'induit chaque processus et la charge que chaque message induit sur les liens de communications.

L'ordonnancement des opérations de calculs et de communications est totalement dynamique, il est pris en charge par le système d'exploitation sous-jacent.

La capacité mémoire nécessaire sur chaque processeur est estimable avant exécution du graphe de processus grace au module *ConfigTool* qui utilise les attributs des processus renseignés lors de la phase de caractérisation. Ensuite il est possible de connaître les valeurs réelles au moyen du module *PerfTool* qui utilise des traces d'exécution générées par des processus "espions" exécutés sur chaque processeur étudié. Ces processus particuliers sont insérés automatiquement par l'outil *ConfigTool* sur les processeurs que l'utilisateur veut étudier.

9.4.4 Génération d'exécutif

Après placement manuel ou automatique, l'outil *ConfigTool* génère un fichier de configuration (*cfsfile*) qui décrit textuellement le graphe de processus, le graphe matériel et le placement, un fichier (*stub*) qui permet d'associer le code de chaque processus avec le contenu du fichier précédent et le makefile qui permet de compiler, linker et charger le code compilé sur les processeurs. Pour chaque processeur il y a génération automatique en langage C des communications (intra-processeur et inter-processeur), du lancement des processus sur chaque processeur et enfin du contrôle du chargement et des espions. L'exécution d'un graphe de processus sur une machine multiprocesseur nécessite la présence d'un système d'exploitation temps-réel supportant l'ordonnancement dynamique des processus et le routage dynamique des communications (machine virtuelle PVM[37] et MPI[92] ou système basé sur les outils Parsytec[81] pour les architectures à base de transputer).

Trapper fournit aussi des outils d'analyse de l'exécution réelle du programme à des fins de déboguage et d'amélioration de performances. Pour cela un système d'"espionnage" basé sur un moniteur (*monitoring system*) est exécuté sur chaque processeur cible. Il permet la collecte d'information transférées pendant ou après l'exécution de l'application vers la machine hôte. Ces informations sont de type événement logiciel ou comportement matériel. Les événements logiciels sont principalement les communications inter-processus et les accès aux variables. Les comportements matériels correspondent à la charge des liens de communications et des processeurs ainsi qu'à l'ordonnancement des tâches. Pour limiter le surcoût de l'espionnage, il est possible de paramétrer ce moniteur afin de lui spécifier uniquement les événements à surveiller (mise en place automatique de processus espions sur certains processeurs). Le moniteur, une fois paramétré, fournit donc des informations étiquetées par une horloge globale (basée sur un mécanisme de synchronisation logiciel). Les informations obtenues peuvent être renvoyées immédiatement vers l'hôte (mode synchrone) ou stockées temporairement dans un buffer (mode asynchrone) pour être renvoyées après exécution vers la machine hôte. Ces informations sont utilisées par l'outil *VisuTool* à des fins de mise au point de l'algorithme et par l'outil *PerfTool* afin d'optimiser la charge de chaque processeur.

VisuTool permet d'étudier graphiquement le comportement du système pendant son exécution (mode "on line") ou après son exécution (mode "off line") à partir des informations fournies par le moniteur. Cet outil est principalement destiné à la mise au point.

PerfTool affiche l'état des composants matériels (charge, processus en cours d'exécution ...) sous la forme d'une animation sur le graphe matériel et d'un diagramme temporel présentant l'évolution des paramètres au cours du temps. Cet outil est essentiellement destiné à l'optimisation.

9.4.5 Conclusion

Cet outil génère une distribution statique et un ordonnancement dynamique. La distribution des communications peut être statique. C'est surtout un outil de mise au point et d'analyse de performances, offrant de nombreuses fonctionnalités d'affichage graphique pour présenter les comportements logiciel et matériel du système. Par contre rien n'indique que cet outil permette de garantir une exécution respectant des contraintes temps réel.

Chapitre 10

Le logiciel SynDEx

Sommaire

Pour répondre spécifiquement à toutes les phases du prototypage rapide d'applications temps réel distribuées embarquées, de la spécification initiale de l'algorithme et de l'architecture jusqu'à l'exécution optimisée temps réel de l'algorithme par les processeurs de l'architecture, nous avons implanté la méthodologie AAA dans un logiciel interactif appelé SynDEx pour **Sync**hronized **D**istributed **Ex**ecutive puisqu'il génère automatiquement un exécutif distribué synchronisé.

Contrairement à CASCH, SynDEx doit supporter la spécification d'architectures hétérogènes et garantir le respect des contraintes temps réel. GEDAE supporte les architectures hétérogènes, mais celles-ci doivent faire partie de bibliothèques qui ne sont pas accessibles à l'utilisateur. SynDEx permet à l'utilisateur de spécifier et d'élargir les architectures qu'il supporte grâce à la modélisation générique basée sur des graphes de la méthodologie AAA. De plus, bien que GEDAE possède des outils puissants de simulation et d'observation de l'exécution temps réel, il ne renferme aucun outil d'optimisation automatique qui reste donc à la charge de l'utilisateur. Un outil de prototypage rapide doit cependant fournir des réponses automatiques à ce type de problème complexe, c'est pourquoi nous avons implanté les heuristiques d'optimisation dans SynDEx. TRAPPER renferme des outils d'optimisation automatique, mais l'ensemble des architectures supportées est, tout comme GEDAE, limité et peu extensible. De plus, cet outil ne fournit aucune garantie sur les performances temps réel du code généré, car comme la plupart des outils que nous avons pu étudier, ce code repose sur un exécutif résidant. SynDEx permet la génération automatique de l'exécutif distribué taillé spécifiquement à l'application. Pour garantir le respect de ces contraintes tout en minimisant les surcoûts de l'exécutif, SynDEx génère automatiquement un exécutif sur mesure, conformément à la seconde partie de cette thèse.

Nous verrons que cette version, en plus d'avoir été entièrement spécifiée et réécrite dans un langage accessible à toute la communauté scientifique, supporte les architectures hétérogènes et effectue une grande

partie des optimisations présentées dans le quatrième chapitre. Le code est généré automatiquement selon les principes exposés dans la seconde partie de cette thèse.

10.1 Présentation

SynDEx est basé sur l'interactivité avec l'utilisateur grâce à une interface graphique qui permet de spécifier le graphe d'algorithme, le graphe d'architecture ainsi que les caractéristiques de ces graphes. SynDEx est ensuite capable de construire et d'afficher automatiquement le diagramme temporel d'implantation optimisé de l'application grâce à l'heuristique qu'il renferme. Si les caractéristiques de ce graphe sont conformes avec les exigences temps réel de l'application, SynDEx est ensuite capable de générer automatiquement l'exécutif de chacun des processeurs de l'applications ainsi que le "makefile" qui va automatiser les différentes phases de substitution, compilation, téléchargement et lancement de l'exécutif de l'application. La figure 10.1 indique ces différentes fonctionnalités offertes par SynDEx et les interactions utilisateur qu'il permet.

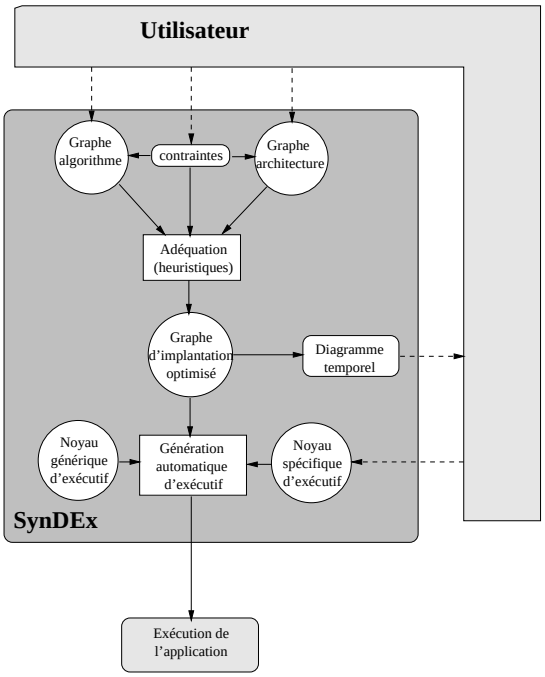


FIG. 10.1: *Fonctionnalités de SynDEx*

10.1.1 Algorithme

L'interface graphique de SynDEx permet donc de spécifier l'algorithme de l'application. Cette spécification correspond exactement au modèle d'algorithme du second chapitre. Les opérations sont représentées par des rectangles. Une opération est connectée à un arc au moyen d'un *port*. Chaque opération possède un ensemble de ports d'entrées et de sorties identifiés par nom unique défini par l'utilisateur. Les noms des ports d'entrée sont préfixés par un point d'interrogation et les noms des ports de sortie par un point d'exclamation. Lors de la génération de code cette liste de ports correspond à la liste d'appel de la macro de l'opération. Chaque sommet du graphe d'algorithme (rectangle) est identifié par un nom et un type choisi par l'utilisateur. Le type de l'opération correspond au nom de la macro qui sera générée par SynDEx. Chaque port de la

liste des ports d'une opération possède aussi un nom et un type. Ce nom de type correspond au type et à la quantité de données associée à la dépendance de données qui sera connectée à ce port (ce sont les deuxième et troisième arguments des quadruplets associés à chaque dépendance du graphe d'algorithme).

Le graphe d'algorithme peut être issu d'une spécification effectuée dans les langages synchrones (Lustre, Esterel, Signal etc) en important une spécification intermédiaire générée en DC[44], le format commun des langages synchrones. SynDEx a aussi été interfacé avec d'autres outils parmi lesquels nous citerons AVS, Scicos et Camlflow.

AVS (Application Visualization System) est un outil de traitement d'image sur station de travail unix. Il permet de décrire et de simuler graphiquement un graphe flot de données de traitement d'image. Le laboratoire ARTIST de l'INSA a développé un traducteur AVS-SynDEX de façon à transformer automatiquement une chaîne de traitement d'image spécifié sous AVS[35] en un graphe d'algorithme qui est ensuite, grâce à SynDEx, automatiquement implanté sur une architecture basée sur des DSP.

Scicos [77] (Scilab Connected Object Simulator) est une boîte à outils du logiciel libre de calcul scientifique Scilab [1]. Scicos contient un éditeur graphique de type schéma-bloc, qui permet la modélisation et la simulation de systèmes dynamiques. Ces systèmes peuvent aussi être hybrides, c'est-à-dire composés à la fois d'éléments évoluant en temps continu, en temps discret (échantillonnés), et (ou) par des activations événementielles (ponctuelles). Ces systèmes sont modélisés par des blocs, représentant des fonctions de base (fournies avec Scicos) ou spécifiques (définies par l'utilisateur). Scicos a été interfacé avec SynDEx[29] de façon à ré-injecter, dans Scicos, les calculs de durée d'exécution effectués par SynDEx, afin d'étudier les conséquences induites par les temps de calcul d'une implantation distribuée. Cela permet d'implanter automatiquement une spécification Scicos sur une machine distribuée, en respectant des contraintes temps réel.

CamlFlow est un logiciel capable de transformer une spécification Caml[18] en une spécification flots de données. Il permet aussi de spécifier textuellement des graphes flots de données. Interfacé avec SynDEx[96], il permet d'implanter rapidement des programmes Caml sur des architectures distribuées.

Exemple 10.1.1 *La figure 10.2 présente une capture d'écran de la version 5 de SynDEx lors du prototypage d'une application très simplifiée de traitement du signal. Le graphe de la moitié inférieure de cette figure correspond au graphe d'algorithme. Les rectangles gensig et visu correspondent respectivement à des opérations d'entrée-sortie. Le rectangle coeff est une opération retard qui permet de transmettre des données entre les itérations de l'algorithme.*

La petite fenêtre, ouverte à droite de l'opération adap indique que le type de cette opération est requal (real equalizer), c'est le nom de la macro qui sera générée automatiquement. La petite fenêtre ouverte sous le port d'entrée "?i" de l'opération visu, indique que ce port est de type fbat.

10.1.2 Architecture

Comme nous l'avons déjà signalé, l'interface graphique de SynDEx permet aussi de spécifier le graphe d'architecture de l'application. Le graphe spécifié ici ne correspond pas exactement au modèle d'architecture présenté dans le premier chapitre de cette thèse mais à une version que l'on qualifiera d'encapsulée. En effet, dans la version 5, chaque rectangle correspond implicitement à un processeur composé d'un unique opérateur, d'un unique média RAM, et d'autant de communicateurs connectés à ce média qu'il y a de portes [2] pour connecter le processeur à des SAM. Les SAM sont représentées par des cercles. Le nom des portes permet d'identifier les communicateurs lors de la génération de code. Chaque processeur (rectangle) du graphe

1. http://www-rocq.inria.fr/scilab

2. Nous avons choisi le qualifi catif portes pour qu'il n'y ait jamais ambiguï t´e avec les ports des op´eration lors des explications

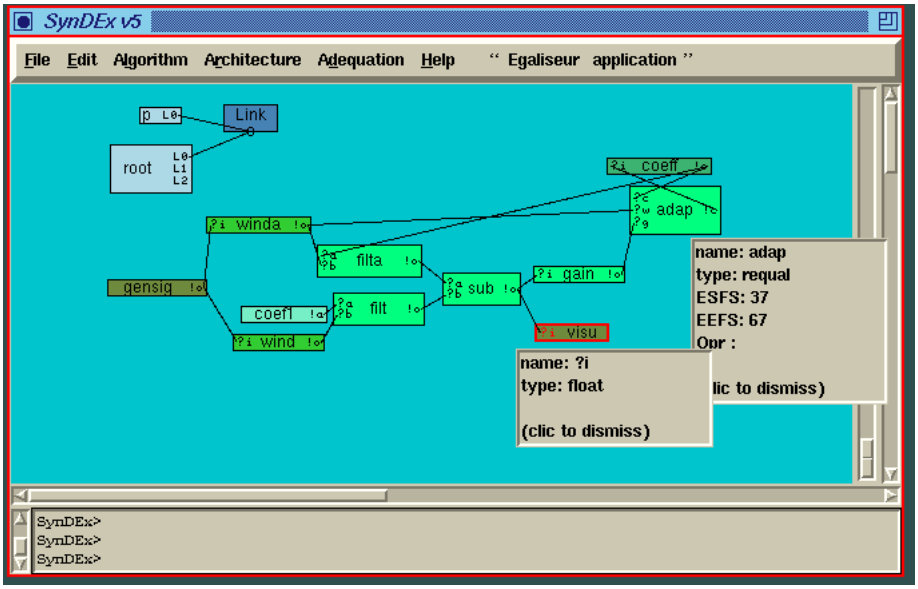


FIG. 10.2: *Spécification d'un algorithme et d'une architecture dans SynDEx V5*

d'architecture est identifié par un nom unique et un nom de type. De même chaque porte (communicateur) est identifiée par un nom de type de porte.

Actuellement cette version ne supporte que les communications par SAM point-à-point ou multipoint avec support matériel du broadcast. Deux autres versions supportant respectivement les communications par SAM multipoint sans diffusion matérielle et les communications par média RAM ont aussi été expérimentées mais ne font pas encore partie de la version 5 que nous distribuons aujourd'hui.

Remarque 50 *Ce modèle d'architecture encapsulé a été longuement utilisé dans la méthodologie AAA[94], [93], [66]. Bien que moins précis (pas de RAM, d'arbitrage) que le modèle présenté dans cette thèse, il a cependant été utilisé avec succès pour implanter de nombreuses applications où l'utilisation des RAM n'était pas critique et les communications par média RAM non nécessaires.*

Exemple 10.1.2 *La partie supérieure de la figure 10.2 présente un graphe d'architecture composée de deux processeurs (rectangles* `root` *et* `p`*) connectés par une mémoire SAM (rectangle* `link` *). Le processeur* `root` *renferme trois communicateurs identifiés par les ports* `L0`, `L1` *et* `L2`. *Le processeur* `p1` *renferme un seul communicateur nommé* `L0`.

10.1.3 Caractérisation

La caractérisation consiste à indiquer pour chaque type de processeur la liste des types d'opérations qu'il est capable d'exécuter, et pour chaque type sa durée d'exécution.

Chaque type de portes (communicateurs) doit être capable de transférer n'importe quel type de données. Donc pour chaque type de porte il faut indiquer la durée de transfert qui correspond à chaque type de données utilisées dans le graphe d'algorithme.

Exemple 10.1.3 *La figure 10.3 est une capture d'écran du formulaire de saisie des caractéristiques de l'application. La colonne de gauche correspond à la liste des types de processeurs définis. La colonne de droite fournit la liste des types d'opérations qui est capable d'exécuter le type de processeur sélectionné. En*

sélectionnant un type d'opération il est possible d'afficher et de modifier sa durée d'exécution pour le type de processeur sélectionné. Dans cet exemple nous constatons que le type de processeur C40 est capable d'exécuter les types d'opération constant, memory, requal, visu etc, et que par exemple le type d'opération requal s'exécute en 30 milli-secondes (l'unité n'est pas imposée par l'outil, il faut simplement donner toutes les durées dans la même unité).

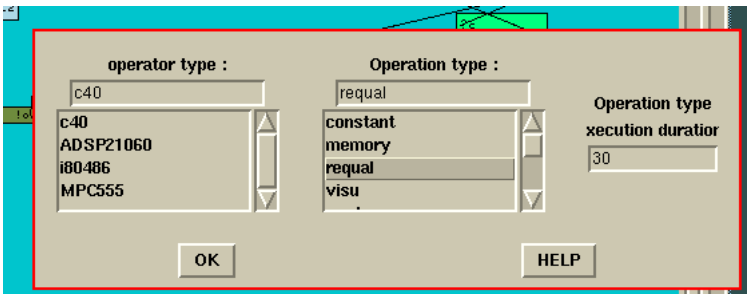


FIG. 10.3: *Spécification des caractéristiques dans SynDEx V5*

10.1.4 Adéquation

Après avoir spécifié et caractérisé les deux graphes de l'application, il suffit de presser un bouton pour exécuter l'heuristique de distribution et d'ordonnancement de SynDEx. La durée de son exécution dépasse rarement quelques minutes. Le graphe d'implantation optimisé et la durée totale d'exécution de l'application sont affichés sous la forme d'un diagramme temporel dans une seconde fenêtre. La liste des processeurs et des SAM y est affichée horizontalement. Étant donné que les communicateurs sont connectés entre eux par des SAM supportant matériellement le broadcast, à chaque opération de communication distribuée sur un communicateur correspond une opération de communication sur les communicateurs connectés (`send`, `receive` ou `sync`). Pour simplifier le diagramme temporel, SynDEx n'affiche donc pas l'ordonnancement des opérations de communication sur chaque communicateur, mais les encapsule dans une seule opération de communication ordonnancée sur la SAM. Pour chacun des processeurs (resp. SAM) SynDEx affiche verticalement les opérations de calcul (resp. communications) distribuées et ordonnancées par l'heuristique. La hauteur des rectangles, qui symbolisent les opérations de calcul ou de communication, est proportionnelle à leur durée d'exécution.

Pour chaque opération il est possible d'afficher les dates d'exécution de chaque opération. Si la durée totale d'exécution ne satisfait pas les contraintes temps réel exigées par l'application, il existe plusieurs façons d'interagir avec l'outil.

Il faut commencer par examiner attentivement la distribution et l'ordonnancement effectuée par SynDEx afin de détecter une éventuelle "faiblesse" de l'heuristique. En effet, si les heuristiques sont capables de donner rapidement des résultats relativement satisfaisant, ce ne sont que des solutions approchées. La prédiction temporelle de l'exécution permet parfois de mettre en évidence ces mauvais choix de l'heuristique. L'interface graphique de SynDEx permet alors à l'utilisateur d'agir sur l'heuristique en imposant des "contraintes de placement" (forcer une opération donnée à être exécutée systématiquement sur un opérateur donné).

Si la prédiction temporelle met en évidence un goulet d'étranglement au niveau des communications, l'utilisateur peut modifier le graphe d'architecture et relancer l'heuristique pour trouver une solution satisfaisante.

Enfin, la prédiction temporelle met parfois en évidence le manque de parallélisme potentiel de l'algorithme vis à vis du parallélisme disponible de l'architecture qui est donc sous-exploité. Dans ce cas l'utilisateur peut essayer de modifier sa spécification puis évaluer l'amélioration de performance en relançant l'heuristique.

L'interactivité est essentielle dans SynDEx. En laissant beaucoup de liberté à l'utilisateur lors de la phase automatique d'optimisation, les possibilités de déterminer rapidement une solution satisfaisante à son problème d'optimisation NP-complet sont accrues.

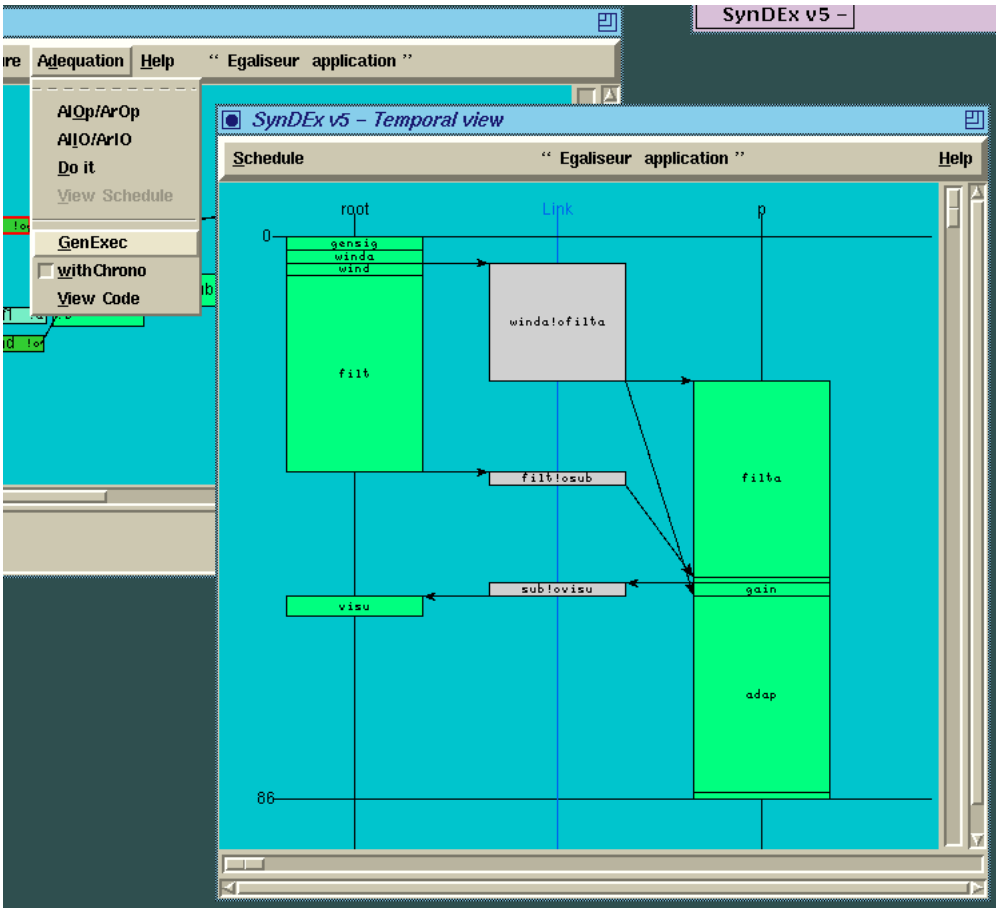


FIG. 10.4: *Résultat de l'adéquation réalisée par SynDEx V5*

Exemple 10.1.4 *La figure 10.4 présente la prédiction temporelle générée par SynDEx pour l'application de la figure 10.2. La colonne de gauche représente l'ordonnancement des opérations distribuées sur le processeur* `root` *et la colonne de droite les opération exécutées par le processeur* `p`*. La colonne centrale donne l'ordonnancement des opérations de communication ajoutées pour transmettre les dépendances de données inter-processeur. La durée d'exécution estimée est de 86 milli-secondes, si cette durée convient, il ne reste plus qu'à générer le code de cette application par SynDEx via le menu "Adéquation → GenExec".*

10.1.5 Génération d'exécutif

Une fois que la prédiction de SynDEx est satisfaisante, il suffit de presser un bouton pour qu'il génère automatiquement le macro-exécutif de chacun des processeurs de l'architecture.

Pour pouvoir transformer ce macro exécutif en exécutif compilable dans le langage de chaque processeur de l'architecture, il faut fournir les noyaux d'exécutif générique de chaque processeur et de chaque type de communicateur. Nous avons vu que ces noyaux sont relativement petits, ainsi nous avons pu développer les noyaux d'un certain nombre de processeurs (TMS320C40, ADSP21060, 68332, MPC555 ...). Le portage d'un noyau à un nouveau type de processeur est relativement rapide, surtout en s'aidant des exécutifs réalisés. Comme nous l'avons expliqué, cet exécutif est indépendant de l'application, il donc réutilisable chaque fois que le même type de processeur fera partie du graphe d'architecture. Rappelons que le nom d'un noyau d'exécutif d'un type de processeur est donné par "NomTypeProc.m4x" et que celui d'un type de communicateur est "NomTypeComm.m4x".

Pour chaque application il peut être nécessaire de fournir un exécutif spécifique à l'application, contenant le code des opérations spécifiques à l'application. Cet exécutif correspond simplement au code de chaque opération spécifique du graphe d'implantation. Rappelons que le nom de ce noyau d'exécutif doit toujours être du type "NomApplication.m4x" pour que l'on puisse automatiser la phase de génération d'exécutif.

Les noyaux d'exécutifs étant définis, il reste simplement à renseigner le type de compilateur de chaque processeur cible dans le makefile avant de taper la commande "make".

Pour chaque fichier de processeur, cette commande va commencer par substituer le macro-code par le code source à partir des noyaux d'exécutifs, puis par déclencher la compilation de chacun d'eux qui vont ensuite être téléchargés (grâce au code des macros `loadOnto _` et `loadFrom _`) puis exécutés dans chacun des processeurs de l'architecture.

Exemple 10.1.5 *Les deux fichiers de la figure 10.5 correspondent au macro-exécutif généré par SynDEx pour les deux processeurs, selon la prédiction temporelle de la figure 10.4.*

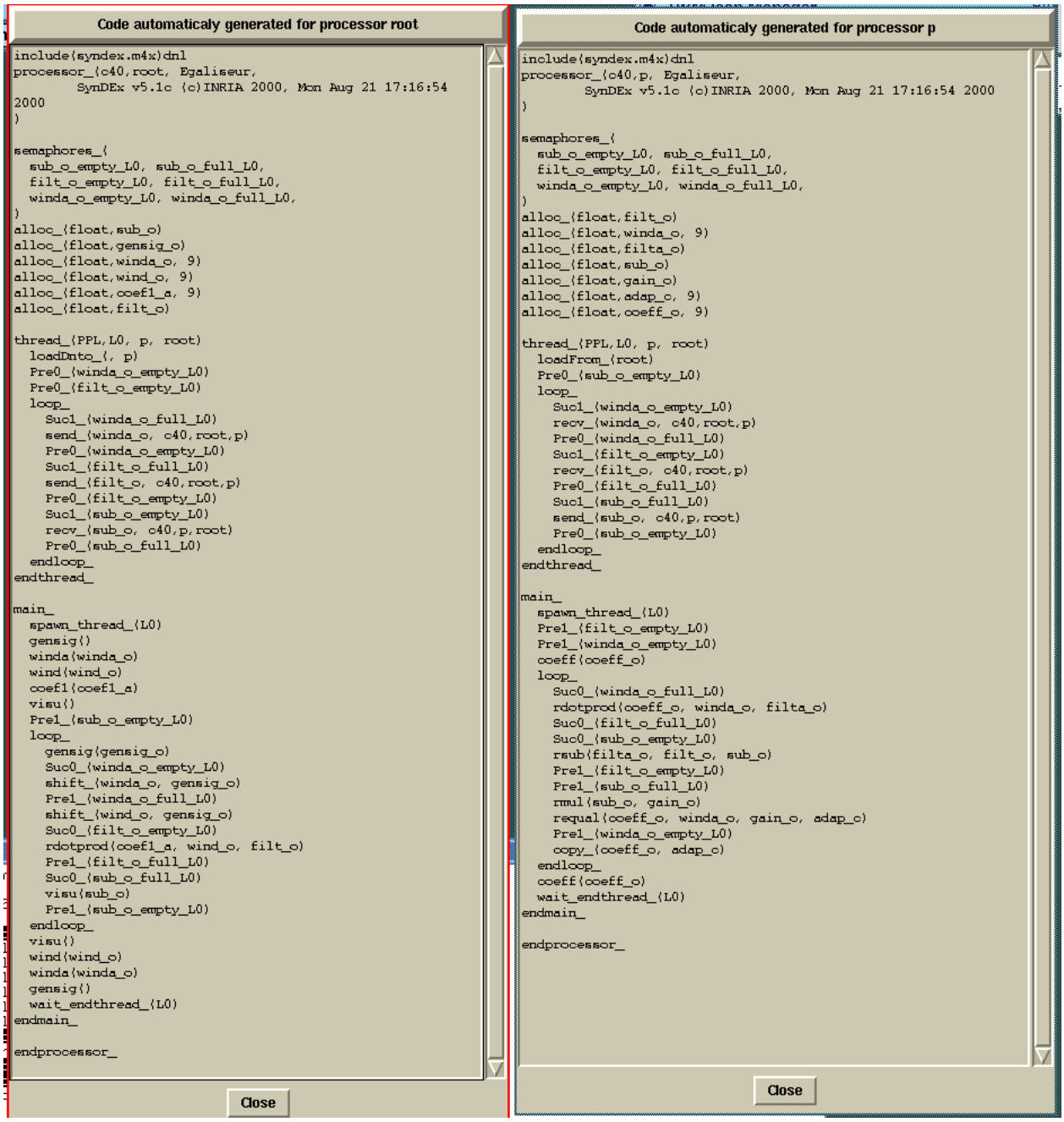
10.2 Structure logiciel de SynDEx

10.2.1 Historique

Les versions précédentes de SynDEx (1 à 4) étaient entièrement développées en Smalltalk[13]. Le choix s'était porté sur ce langage pour plusieurs raisons ; c'est un langage orienté objet accompagné d'un ensemble d'outils intégrés qui permettent d'interagir avec les composants de ce langage. Il possède tous les outils classiques des environnements de programmation, compilateur, deboggeur, éditeur de texte. Smalltalk intègre aussi toutes les bibliothèques graphiques nécessaires pour exploiter les environnements graphiques de type X11, Windows9x, Windows NT, etc. Le code qu'il génère est interprété par une machine virtuelle qui existe sur la plupart des environnements actuels (X11, Windows, Macintosh), l'application créée en Smalltalk est donc portable, sans modification des sources, sur un grand nombre de plates-formes.

C'est pour l'ensemble de ces raisons que le langage Smalltalk a été retenu pour la programmation de SynDEx. Ses avantages ont permis de réduire les temps de développement logiciel, l'équipe de recherche a donc pu expérimenter et affiner beaucoup plus rapidement la théorie développée dans la méthodologie A^3, sans avoir besoin d'écrire de nombreuses spécifications comme l'exigent d'autres langages moins souples.

Cependant, ce choix de langage n'est pas sans conséquence, bien que les avantages de ce langage restent d'actualité, il est aujourd'hui peu implanté dans le monde de la programmation, il existe peu de programmeurs Smalltalk dans le contexte du temps réel qui nous concerne, tant dans le monde de la recherche que dans celui du monde industriel. Smalltalk est surtout utilisé aujourd'hui pour réaliser des applications de gestion de base de données.



Code automaticaly generated for processor root

```
include(syndex.m4x)dnl
processor_(c40,root, Egaliseur,
        SynDEx v5.1c (c)INRIA 2000, Mon Aug 21 17:16:54
2000
)

semaphores_(
  sub_o_empty_L0, sub_o_full_L0,
  filt_o_empty_L0, filt_o_full_L0,
  winda_o_empty_L0, winda_o_full_L0,
)
alloc_(float,sub_o)
alloc_(float,gensig_o)
alloc_(float,winda_o, 9)
alloc_(float,wind_o, 9)
alloc_(float,coef1_a, 9)
alloc_(float,filt_o)

thread_(PPL,L0, p, root)
  loadDnto_(, p)
  Pre0_(winda_o_empty_L0)
  Pre0_(filt_o_empty_L0)
  loop_
    Suc1_(winda_o_full_L0)
    send_(winda_o, c40,root,p)
    Pre0_(winda_o_empty_L0)
    Suc1_(filt_o_full_L0)
    send_(filt_o, c40,root,p)
    Pre0_(filt_o_empty_L0)
    Suc1_(sub_o_empty_L0)
    recv_(sub_o, c40,p,root)
    Pre0_(sub_o_full_L0)
  endloop_
endthread_

main_
  spawn_thread_(L0)
  gensig()
  winda(winda_o)
  wind(wind_o)
  coef1(coef1_a)
  visu()
  Pre1_(sub_o_empty_L0)
  loop_
    gensig(gensig_o)
    Suc0_(winda_o_empty_L0)
    shift_(winda_o, gensig_o)
    Pre1_(winda_o_full_L0)
    shift_(wind_o, gensig_o)
    Suc0_(filt_o_empty_L0)
    rdotprod(coef1_a, wind_o, filt_o)
    Pre1_(filt_o_full_L0)
    Suc0_(sub_o_full_L0)
    visu(sub_o)
    Pre1_(sub_o_empty_L0)
  endloop_
  visu()
  wind(wind_o)
  winda(winda_o)
  gensig()
  wait_endthread_(L0)
endmain_

endprocessor_
```

Close

Code automaticaly generated for processor p

```
include(syndex.m4x)dnl
processor_(c40,p, Egaliseur,
        SynDEx v5.1c (c)INRIA 2000, Mon Aug 21 17:16:54 2000
)

semaphores_(
  sub_o_empty_L0, sub_o_full_L0,
  filt_o_empty_L0, filt_o_full_L0,
  winda_o_empty_L0, winda_o_full_L0,
)
alloc_(float,filt_o)
alloc_(float,winda_o, 9)
alloc_(float,filta_o)
alloc_(float,sub_o)
alloc_(float,gain_o)
alloc_(float,adap_c, 9)
alloc_(float,coeff_o, 9)

thread_(PPL,L0, p, root)
  loadFrom_(root)
  Pre0_(sub_o_empty_L0)
  loop_
    Suc1_(winda_o_empty_L0)
    recv_(winda_o, c40,root,p)
    Pre0_(winda_o_full_L0)
    Suc1_(filt_o_empty_L0)
    recv_(filt_o, c40,root,p)
    Pre0_(filt_o_full_L0)
    Suc1_(sub_o_full_L0)
    send_(sub_o, c40,p,root)
    Pre0_(sub_o_empty_L0)
  endloop_
endthread_

main_
  spawn_thread_(L0)
  Pre1_(filt_o_empty_L0)
  Pre1_(winda_o_empty_L0)
  coeff(coeff_o)
  loop_
    Suc0_(winda_o_full_L0)
    rdotprod(coeff_o, winda_o, filta_o)
    Suc0_(filt_o_full_L0)
    Suc0_(sub_o_empty_L0)
    rsub(filta_o, filt_o, sub_o)
    Pre1_(filt_o_empty_L0)
    Pre1_(sub_o_full_L0)
    rmul(sub_o, gain_o)
    requal(coeff_o, winda_o, gain_o, adap_c)
    Pre1_(winda_o_empty_L0)
    copy_(coeff_o, adap_c)
  endloop_
  coeff(coeff_o)
  wait_endthread_(L0)
endmain_

endprocessor_
```

Close

FIG. 10.5: *Fichiers de macro-exécutif générés par SynDEx V5*

10.2.2 Structure

Pour permettre à un plus grand nombre d'utilisateurs de pouvoir adapter SynDEx à leurs besoins spécifiques (expérimentations de nouvelles heuristiques [102],), gestion des graphes de contrôle pour les automaticiens [56], ajout de la tolérance aux pannes[38] [3] tout en apportant les nouvelles fonctionnalités déve-

3. Collaboration entre le projet SOSSO de l'INRIA Rocquencourt et le projet BIP (INRIA Rhˆone-Alpes) `a travers l'action de recherche coop´erative TOLÈRE sur les tol´erance aux fautes pour les syst`emes temps r´eel r´epartis

loppées dans cette thèse, nous avons choisi de redévelopper la version 5 de SynDEx dans un langage plus répandu en y ajoutant de nouvelle fonctionnalités. Notre choix s'est porté sur le langage orienté objet C++ [97] qui présente l'intérêt d'être largement enseigné mais aussi d'exister sur tous les types de plate-forme actuelles (Unix, Windows Macintosh). Nous avons séparé la partie graphique afin de la développer dans le langage Tcl/Tk [79, 45] qui présente l'avantage d'être porté sur toutes ces plate-formes, tout en étant à la fois simple et suffisamment puissant pour manipuler des graphes de façon interactive.

SynDEx V5 est donc basé sur deux parties qui interagissent : le cœur, écrit en C++ et l'interface homme machine (IHM) en Tcl/Tk (Cf. figure 10.6).

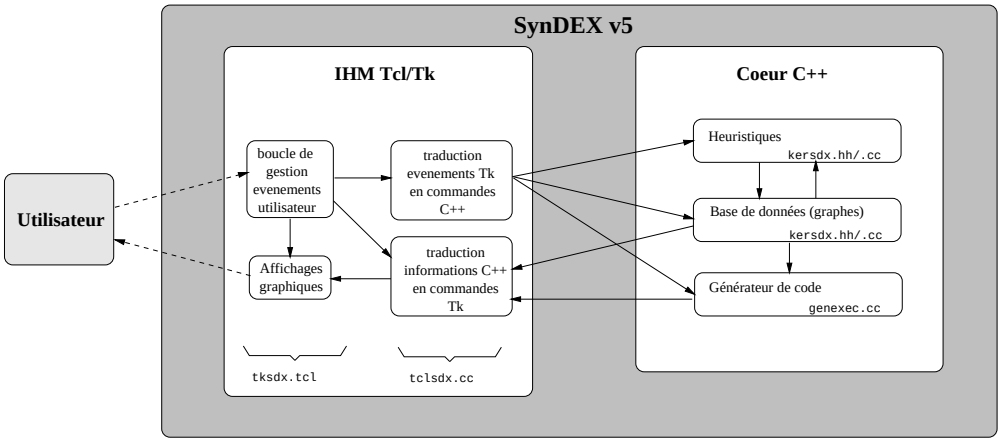


FIG. 10.6: *Architecture fonctionnelle de SynDEx*

10.2.2.1 Cœur

Le cœur C++ de SynDEx contient la base de données de l'application (graphes d'algorithme, d'architecture) les heuristiques d'optimisation (distribution/ordonnancement) ainsi que le générateur automatique d'exécutif. Pour ouvrir le cœur et permettre l'extension de ces fonctionnalités par la communauté scientifique, sa programmation a été précédée d'une phase de spécification détaillée (car il n'en existait pas pour les version Smalltalk précédentes) qui a fait l'objet d'un rapport de DEA [40]. Ce dernier est en partie basé sur le formalisme OMT[88] qui permet de spécifier l'implantation de SynDEx de manière orienté objet.

La programmation orientée objet [4] a été introduite pour permettre une programmation modulaire afin de maîtriser la complexité croissante des logiciels. Pour cela elle est basée sur un mécanisme d'encapsulation qui vise à la protection de données contenues dans des classes qui servent elles mêmes à générer des objets :

un **objet** regroupe un jeu d'opération et un état qui mémorise le résultat de ces opérations. Les objets communiquent entre-eux par envoi de **messages** qui invoquent ces opérations.

une **classe** est une entité génératrice d'une famille d'objets -ses **instances**- dont elle définit la structure et le comportement par les propriétés relationnelles -ses **attributs**-, et fonctionnelles -ses **méthodes**-, qui lui sont associées. Une classe peut être définie à partir d'une ou plusieurs autres classes **ancêtres** par le mécanisme d'**héritage**. Une nouvelle instance d'une classe peut être définie par le mécanisme d'instanciation. Les méthodes de la classe instanciée existeront donc pour toutes les instances de la classe et des classes dont la classe est un ancêtre.

4. POO

La communication par envoi de message est la seule structure de contrôle des langages orientés objets. Elle repose sur le principe fondamental selon lequel l'objet expéditeur d'un message n'a connaissance que du seul sélecteur du message, la propriété associée à ce sélecteur ne dépendant que de l'objet receveur du message. Ainsi à un message de sélecteur donné, il pourra être associé plusieurs propriétés en fonction de l'objet qui reçoit le message. Cela veut donc dire qu'il est possible de définir plusieurs propriétés de même nom sur des classes différentes, on parle alors de **polymorphisme**.

OMT est une Technique de Modélisation par Objet (Object Modeling Technique) employée pour analyser les spécification d'un problème, en concevoir une solution, et implémenter cette solution avec un langage de programmation orienté objet. Cette méthode repose sur trois modèles pour décrire le système à concevoir : le mod èle objet décrivant les objets et leurs relations dans le système; le mod èle dynamique décrivant les interactions entre les objets dans le système; le mod èle fonctionnel décrivant la transformation des données dans le système. Le modèle objet décrit la structure statique des objets (identité, attributs, opérations) et leurs relations avec les autres objets. Il contient des diagrammes d'objets. Un diagramme d'objets est un graphe dont les noeuds sont les classes d'objets et dont les arcs sont les relations entre ces classes. Le modèle dynamique décrit les aspects du système qui se modifient avec le temps. Le modèle dynamique s'emploie pour spécifier et implémenter les aspects contrôle du système. Il contient des diagramme d'états. Un diagramme d'états est un graphe dont les noeuds sont les états et dont les arcs sont les transitions entre les classes provoquées par des évènements. Le modèle fonctionnel décrit la transformation des valeurs des données dans le système. Le modèle fonctionnel contient des diagrammes à flots de données. Un diagramme à flots de données représente un calcul. C'est un graphe dont les noeuds sont les traitements et dont les arcs sont les flots de données

Voici l'ensemble des classes que nous avons retenue pour construire le noyau de SynDEx, la figure 10.7 donne les relations entre ces classes en utilisant le modèle objet d'OMT. Les figures 10.8 et 10.9 à la modèlisation dynamique et à la modélisation fonctionnelle de l'heuristique d'optimisation de SynDEx.

- Applications : c'est la classe à laquelle sont rattachées les instances des classes graphe logiciel et graphe d'architecture qui composent l'application. C'est en effet par l'intermédiaire de cette classe que l'on peut associer les éléments qui composent les deux graphes,
- Graphe _logiciel : cette classe possède les liens qui permettent d'accéder aux objets de la classe opération qui forment le graphe logiciel. Il n'existe pas de classe pour représenter et symboliser les arcs du graphe logiciel qui correspondent simplement à l'association de ports logiciels. C'est avec cette classe que l'on peut créer, connecter, supprimer les opérations,
- Op ération : tous les sommets du graphe matériel sont des instances de cette classe. Les propriétés d'une opération sont définies dans cette classe, les objets de la classe port logiciel sont reliés à cette classe capable de gérer directement toutes les dates qui bornent l'exécution de l'opération,
- Type _op ération : cette classe permet d'implanter toutes les caractéristiques de durée des objets de la classe opération,
- Port _logiciel : cette classe a été créée pour représenter les ports logiciels de chaque opération. Un port logiciel sortant peut être connecté à un ou plusieurs ports entrant, de ce fait, nous avons choisi de tirer parti de cette différence entre les ports entrant et sortant, pour définir que ce sont les ports sortant d'une opération qui tiendront la liste des ports entrant connectés selon les arcs du graphe logiciel. Ainsi il n'a pas été nécessaire de créer une classe connexion pour symboliser ces arcs du graphe logiciel. Nous aurons deux classes héritières de cette classe : la classe port logiciel entrant et la classe port logiciel sortant,

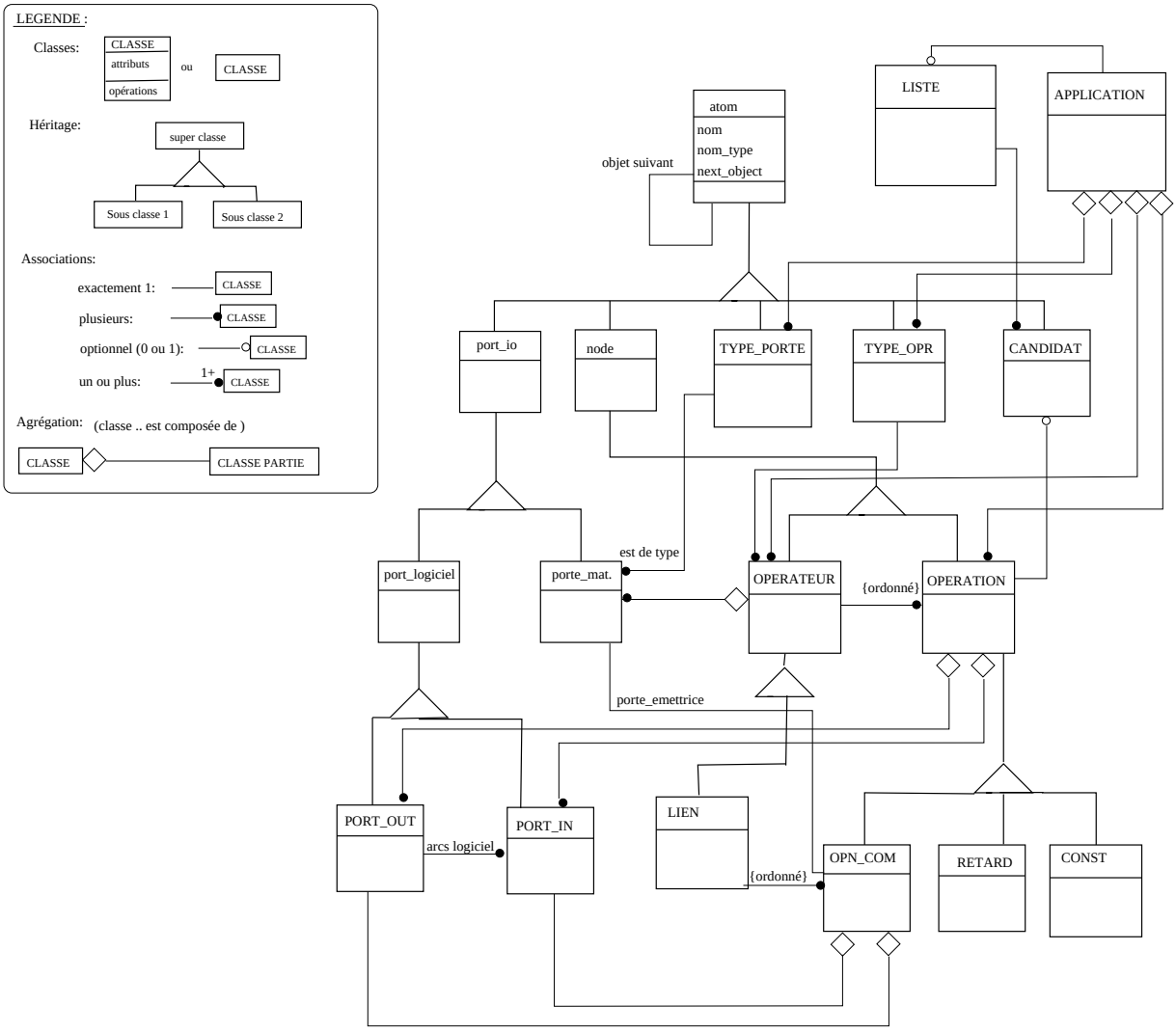


FIG. 10.7: *Modèle objet de SynDEx V5*

– `Type de port_logiciel` : cette classe est associée à la précédente pour l'identifier et implanter toutes les caractéristiques du port, elle contient ainsi toutes les informations sur les données transportées par le port logiciel,

– `Graphe_architecture` : cette classe possède les liens qui permettent d'accéder aux objets de la classe opérateur qui constituent le graphe d'architecture. Contrairement au cas de la classe graphe logiciel, les arcs du graphe matériel sont ici représentés par des objets de la classe liens. C'est avec cette classe que l'on peut créer, connecter, supprimer les opérateurs.

– `Opérateur` : tous les sommets du graphe matériel sont des instances de cette classe. Chaque opérateur possède des portes (communicateurs) qui sont des objets reliés par cette classe. C'est à partir de cette classe que l'on construit, modifie, détruit, lie des portes matérielles,

– `Type_opérateur` : permet de définir toutes les caractéristiques particulières d'un objet de la classe opérateur,

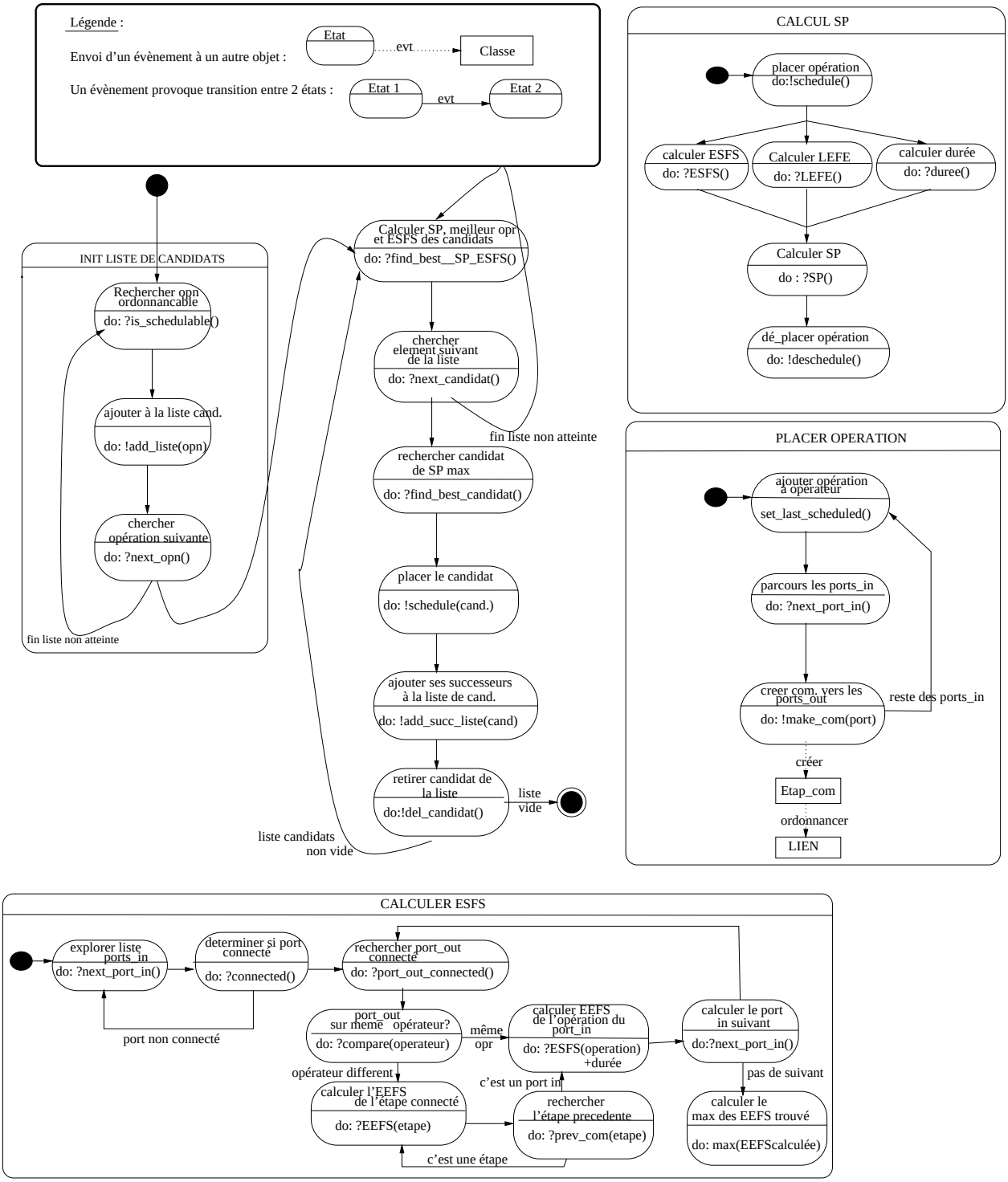


FIG. 10.8: *Modèle dynamique de l'heuristique*

– `Porte` : les instances de cette classe représentent les portes de communication de l'objet opérateur. Les portes de même type mais d'opérateurs différents peuvent être connectées entre elles et forment

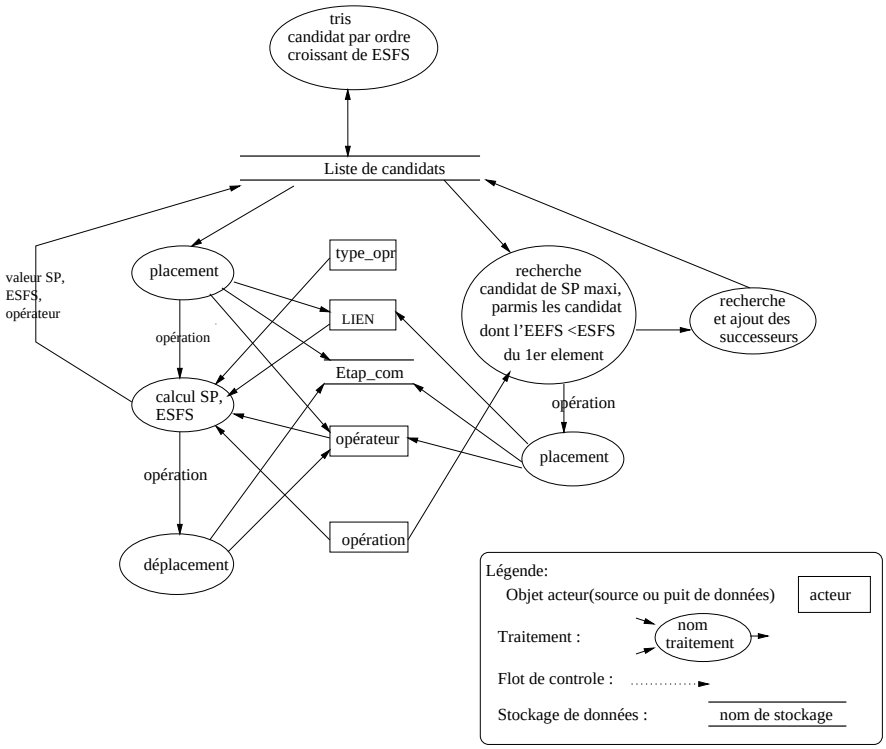


FIG. 10.9: *Modèle fonctionnel de l'heuristique*

un lien (il correspond au média SAM). Un objet de la classe lien leur est donc associé,

– `Lien` : cet objet est créé lorsque l'on connecte des portes de même type mais d'opérateurs différents entre elles. Cette classe permettra de mémoriser ce qui a trait aux communications entre opérateurs par l'intermédiaire d'objets de la classe étape_communication,

– `Étape_communication` : les instances de cette classe représentent les opérations de communications placées sur un lien (distribués sur les communicateurs connectés par le lien SAM),

– `Type_porte_matérielle` : permet d'associer toutes les caractéristiques propres d'une porte,

– `Itérateur` : cette classe particulière sera utilisée pour parcourir les listes d'objets associés à un objet. Par exemple pour parcourir les opérations du graphe logiciel, les opérateurs du graphe matériel, les ports logiciels d'une opération, les portes d'un opérateur, etc,

– `Liste de candidats` : cette classe a été spécialement conçue pour simplifier la programmation de l'heuristique, elle permet de mémoriser la liste d'opération candidate au placement à chaque étape de l'heuristique.

10.2.2.2 IHM

L'IHM, développée en Tcl/Tk fournit une interface graphique utilisateur pour manipuler le cœur C++ de SynDEx. Elle permet à l'utilisateur de spécifier graphiquement les graphes d'algorithme et d'architecture,

de lancer et d'interagir avec l'heuristique en permettant de spécifier des contraintes de distribution, d'afficher graphiquement le graphe d'implantation sous la forme d'un diagramme temporel et enfin de lancer la génération automatique de code.

La figure 10.6 présente les relations entre l'IHM et le cœur C++ ainsi que les différents fichiers mis en jeux. L'IHM est basée sur une boucle d'attente et de gestion des évènements produits par l'utilisateur (clic souris, sélection menu etc). A chaque événement est associée une commande qui va agir sur la base de données C++ (construction des graphes, exécution de l'heuristique etc). En réponse à certaines commandes, le cœur C++ retourne des informations de la base C++ qui sont traduites sous forme graphique par l'IHM. C'est par exemple le cas du diagramme temporel qui est construit en explorant la liste des opérations distribuées sur chaque processeur de l'architecture.

Chapitre 11

Applications de SynDEx

Sommaire

Ce chapitre présente deux exemples d'applications développées à l'aide de la méthodologie AAA et du logiciel SynDEx.

11.1 PROMPT, projet RNTL

11.1.1 Présentation

PROMPT [6][3] (Placement Rapide Optimisé sur Machines Parallèles pour applications Télécoms) est un projet du *R´eseau National de Recherche en T´el´ecommunications* (RNRT) qui réunit Thomson-CSF Communications, Thomson-CSF Laboratoire Central de Recherches (LCR), ARMINES Centre de Recherche en Informatique, SIMULOG et l'INRIA Rocquencourt. Il a pour but de définir une démarche de conception pour les applications temps réel (telles que les traitements de type antenne intelligente pour l'UMTS et le traitement Front-End pour les stations de base en télécommunications) implantées sur des *System On a Chip* (Cf. § 1.1.1.6, p. 6) hétérogènes qui intègrent à la fois des unités de calcul de type SIMD, des unités de calcul flottantes et des processeurs d'usage général. En effet, il n'existe pas d'outils permettant de développer de façon rapide et aisée les applicatifs s'exécutant sur des SOC, tant au niveau du placement optimisé des données sur les structures SIMD qu'au niveau de la distribution et de l'ordonnancement optimisés des opérations de l'algorithme sur la structure MIMD correspondant aux autres processeurs que le SIMD, et qu'au niveau de la génération d'exécutif pour les communications inter-unités.

Si la méthodologie AAA, développée dans cette thèse, est bien adaptée aux architectures homogènes ou hétérogènes de type MIMD, elle ne traite pas spécifiquement de l'optimisation du placement des données dans les unités SIMD comme le fait l'approche PLC2 développée par le LCR. Cette approche est en effet spécialisée dans l'implantation d'application de traitement du signal systématique (à base de nids de boucles parfaitement imbriquées) sur des architectures homogènes de type SIMD. La PLC2, implantée dans

le logiciel EPHORAT[4], est basée sur le traitement global de façon concurrente de quatre fonctions (le partitionnement, l'alignement, la distribution et le séquencement) en utilisant la programmation par contrainte [43].

L'objectif de ce projet consiste à fournir un prototype d'environnement de développement pour les SOC fondé sur la coopération de la méthodologie AAA et de l'approche PLC2, et plus précisément par le couplage de leur environnement logiciel respectif, SynDEx et EPHORAT.

11.1.2 Architecture

Description

Dans le cadre de cette thèse, nous nous sommes plus particulièrement intéressés à la modélisation des SOC à travers un exemple concret fourni par Thomson-CSF Communications dans le cadre de ce projet. Ce SOC, appelé MEFISTO, regroupe sur une même puce de silicium, une unité de calcul SIMD (*Marañon*), une unité de traitement de signal flottant (*mAgic-FPU*) et un processeur d'usage général (*ARM*). Ce SOC possède des capacités de connexion en structure multi-SOC permettant de très fortes puissances de calcul. La figure 11.1 présente les différentes unités de ce SOC ainsi que leurs interconnexions.

- Le Marañon est une unité SIMD basée sur 4 unités Piranha complètement connectées, chacune capable de fonctionner à 100Mhz. Chaque Piranha repose sur une architecture VLIW organisée autour de plusieurs bancs mémoire et de quatre unités arithmétiques capables d'effectuer chacune en parallèle des opérations complexes telle que multiplication-accumulation. Le Marañon est capable de délivrer une puissance totale de 400MAC (Multiplication ACcumultation) par seconde. Il communique avec l'extérieur au moyen de mémoires double port d'une part et de FIFOs d'autre part. Les mémoires double ports sont connectées à des périphériques d'entrée de type convertisseur analogique numérique alors que les FIFO permettent l'échange bidirectionnel de données intra-SOC entre le Marañon et le mAgic-FPU;

- Le mAgic-FPU travaille sur des nombres complexes, ses unités traitent en parallèle les parties réelles (*Left*) et imaginaires (*Right*). Il fonctionne à une fréquence de 100MHz, il est basé sur un séquenceur de calcul qui travaille sur un double bloc de mémoire (page0 R et page0 L) de 1kilo-mots de 40 bits chacun. Il peut échanger des données avec le Marañon et une mémoire externe de 16 Méga mots complexes de 80 bits par l'intermédiaire d'un mécanisme de permutation de pages : pendant qu'il accède à l'une des deux pages, la seconde page est adressée en lecture ou en écriture par un double DMA (LAGU et RAGU) à deux canaux (pour la lecture et pour l'écriture). Ce dernier réalise les transferts entre le double bloc mémoire (page1 R et page1 L) et les FIFO de communication avec les Marañon. Il peut aussi réaliser des transferts avec la mémoire externe à l'aide d'un autre DMA ($XAGU$) dédié à l'adressage de cette mémoire. Enfin, le mAgic-FPU peut aussi communiquer avec l'ARM au moyen d'un registre interne dédié et de deux FIFOs (une pour chaque sens);

- L'ARM implanté dans Mefisto (ARM7 TDMI) fonctionne à 50MHz (la moitié de la fréquence des deux autres unités). Il est connecté à un certain nombre de périphériques embarqués sur le SOC tels qu'un timer, un convertisseur analogique-numérique/numérique-analogique, un watchdog et un DMA. Ce dernier permet de transférer des données entre une mémoire de masse externe et les FIFO du mAgic-FPU, ainsi qu'entre cette mémoire externe et la mémoire programme du mAgic-FPU. Il ne peut pas échanger directement des données avec le Marañon;

- Synchronisations inter-unités : l'ARM peut piloter le démarrage du Marañon et du mAgic-FPU au moyen des signaux de contrôle `Start Frame` . Il est informé de l'état du Marañon et du mAgic-FPU au moyen des signaux `End Frame` et `flag` . Il est informé de l'état des FIFO de communication par des signaux d'interruption classiques de types `FIFO vide` et `FIFO pleine` .

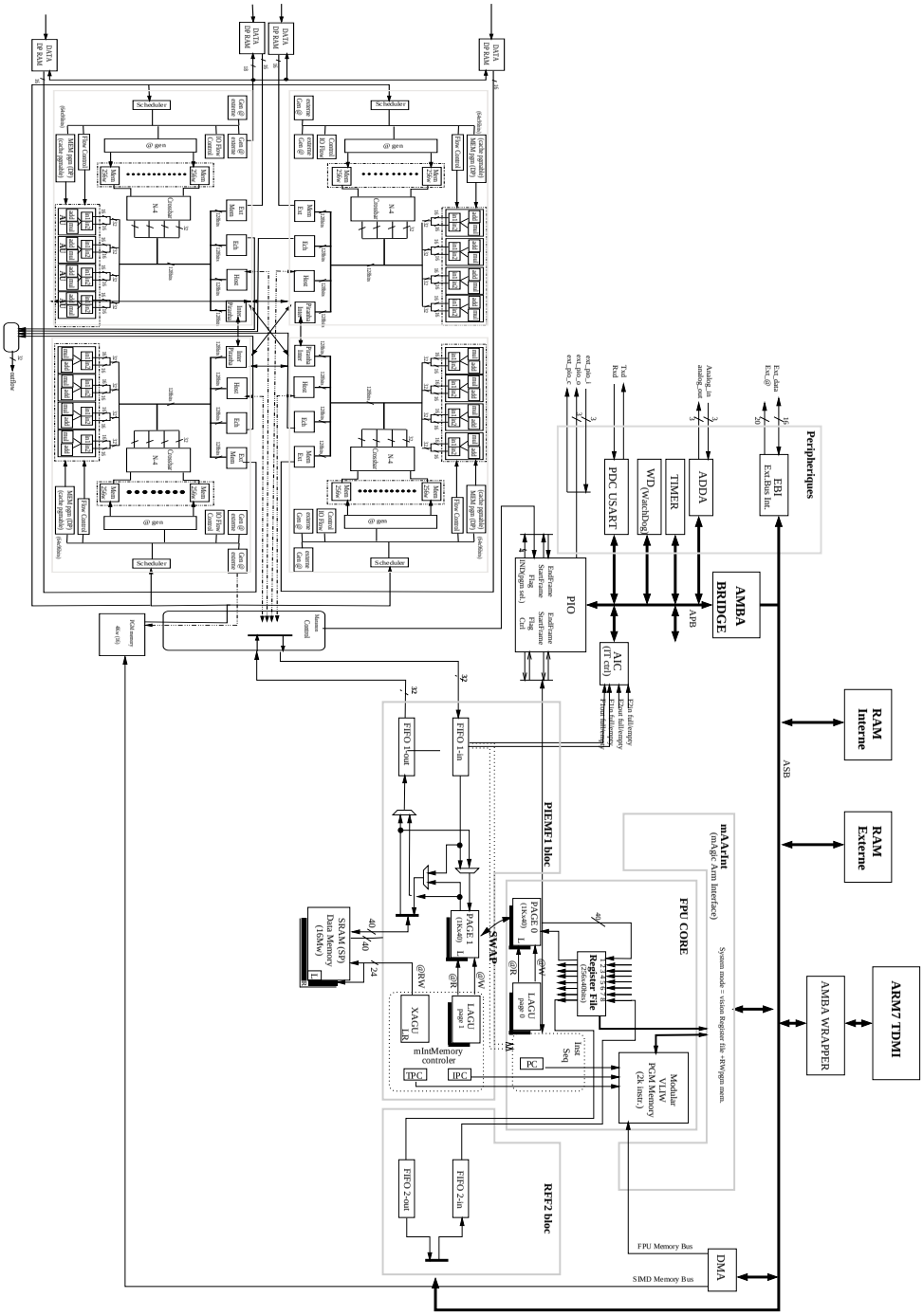


FIG. 11.1: *Architecture interne du SOC "Mefisto"*

Modélisation

Étant donné que les optimisations de placement intra-Marañon seront réalisées par la PLC2, il est inutile, au niveau de la méthodologie AAA, de modéliser toutes ses unités internes. Nous modélisons donc le Marañon par un unique opérateur connecté à une mémoire RAM. Le contenu de cette dernière est transféré par le communicateur connecté à deux mémoires SAM ($S1$ et $S2$ sur la figure11.2). Les FIFO entre l'ARM et le mAgic-FPU sont modélisées par les les SAM $S3$ et $S4$. Le séquenceur de calcul du mAgic-FPU est modélisé par un opérateur, comme celui de l'ARM. Chaque DMA du mAgic-FPU est modélisé par un communicateur comme indiqué sur la figure 11.2.

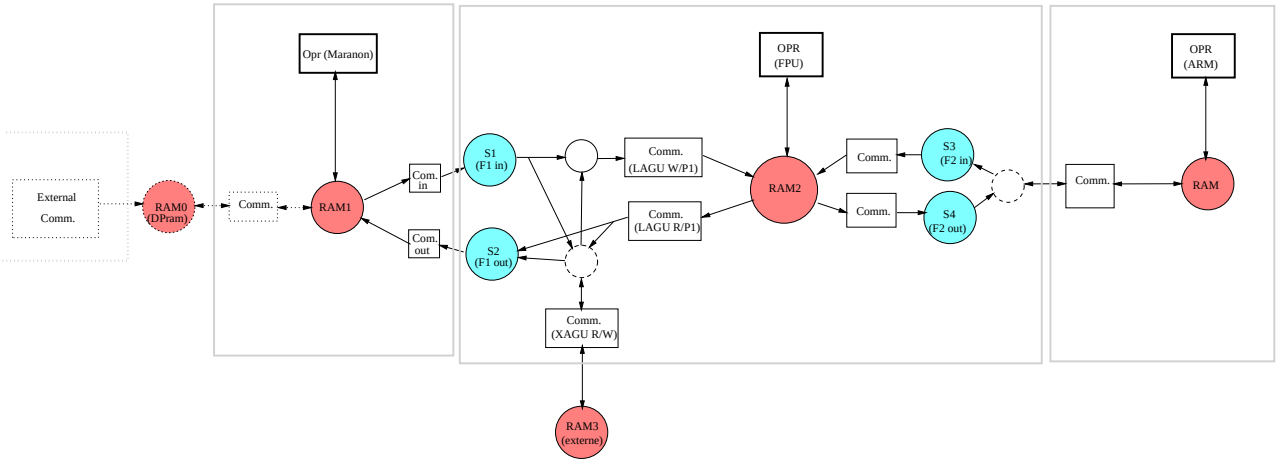


FIG. 11.2: *Modélisation de l'architecture du SOC "Mefisto"*

11.1.3 Couplage SynDEx/EPHORAT

Pour prendre en compte simultanément les aspects réguliers et irréguliers des algorithmes de télécommunications ainsi que les aspects à la fois SIMD et MIMD requit dans les SOC utilisés en télécommunications, nous avons couplé les logiciels SynDEx et EPHORAT. Dans ce couplage, c'est toujours SynDEx qui effectue la distribution et l'ordonnonancement des opérations de calcul et de communication sur les différents opérateurs et communicateurs qui modélisent le SOC. Cependant, lorsqu'une opération de calcul est distribuée sur un opérateur de type SIMD, EPHORAT est utilisé pour d'une part fournir un placement optimisé des données dans le SIMD, et d'autre part pour fournir la durée d'exécution de cette opération de calcul. Le générateur d'exécutif de SynDEX est utilisé pour la partie MIMD et le générateur d'EPHORAT pour la partie SIMD.

Pour étudier concrètement ce couplage, Thomson-CSF nous a fournit des algorithmes de traitement du signal utilisés dans les stations de bases de télécommunications. Nous présentons l'implantation avec SynDEx d'un algorithme "d'entrée en réseau" (Cf. haut de la figure 11.3) sur le Marañon et le mAgic-FPU du SOC Mefisto (Cf. haut gauche de la figure 11.3). Cet algorithme est composé d'opérations de calcul `filtPuiCor`, `detect`, `max`, `filt_final`, `Corr` et `pond` , chacune basée sur des nids de boucles parfaitement imbriquées. Le capteur est modélisé par une opération d'entrée-sortie appelée `antennes` et l'actionneur est modélisé par une opération d'entrée-sortie appelée `modem` . Le capteur produit des données consommées par `filtPuiCor` et `Corr` qui, à leur tour produisent des données pour `detect` et `cond` et ainsi de suite jusqu'à être consommées par l'opération de sortie `modem` . Le mAgic-FPU (modélisé par le sommet de même nom sur la figure 11.3) étant un processeur DSP à virgule flottante, il

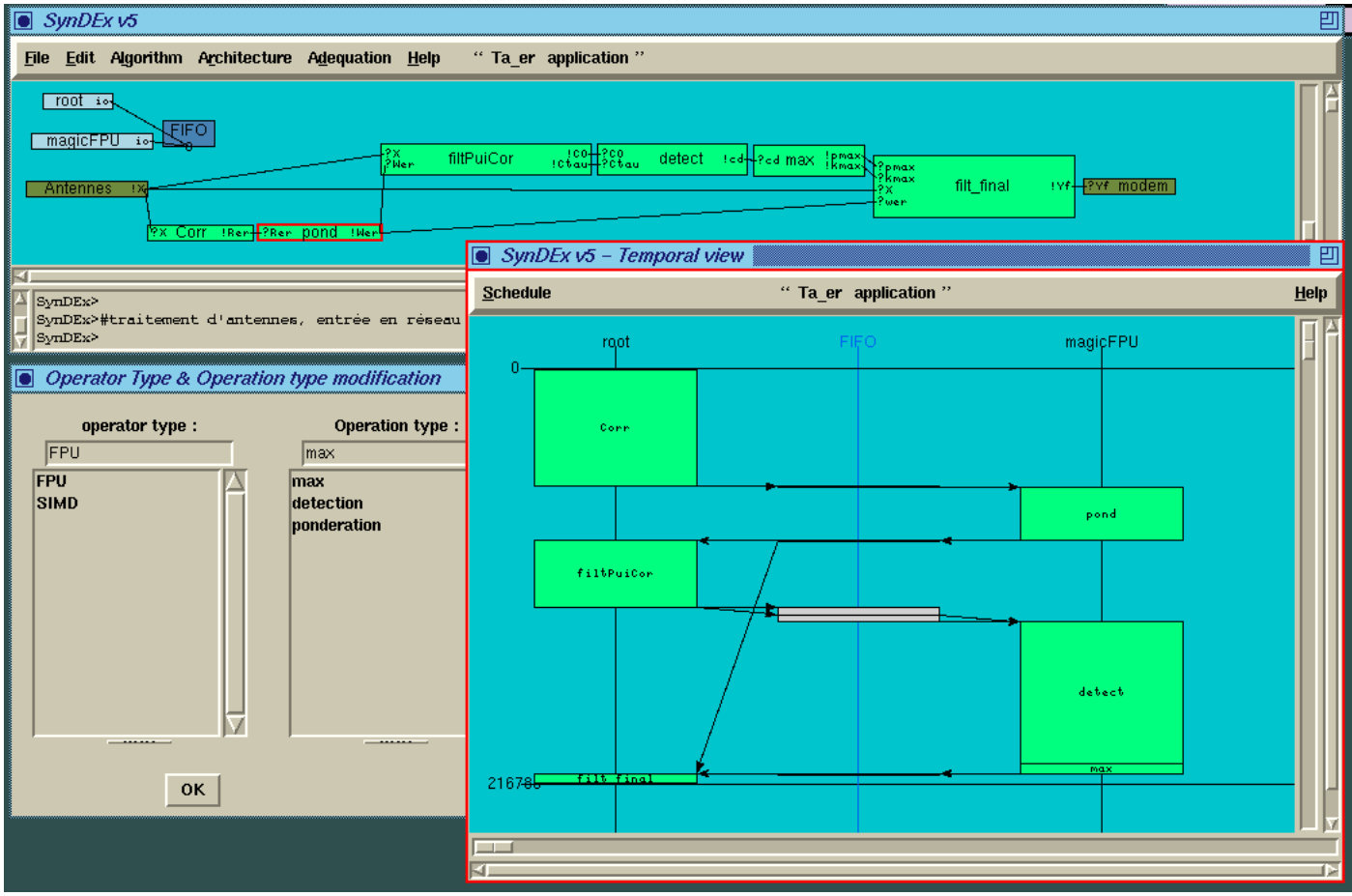


FIG. 11.3: *SynDEx : Implantation d'un algorithme de télécommunications*

est capable d'exécuter toutes les opérations de calcul de l'algorithme. Cependant, les opérations de calcul régulières manipulant des nombres non flottants seront exécutées plus lentement que sur le Marañon (modélisé par le sommet nommé "root" sur la figure 11.3). Ce dernier est uniquement capable d'exécuter des opérations manipulant des nombres non flottants. `Corr`, `filtPuiCor` et `filt_final` opèrent sur des nombres non flottants et requièrent une forte puissance de calcul, le Marañon est donc plus adapté à leur exécution. En revanche, toutes les autres opérations du graphe d'algorithme opèrent sur des nombres flottants, elles sont donc uniquement exécutables par le mAgic-FPU. Comme nous l'avons indiqué précédemment, le logiciel EPHORAT est utilisé pour effectuer un placement des données et calculer la durée d'exécution des opérations ordonnancées sur le Marañon. La figure 11.4 présente les résultats obtenus pour le sommet `filtpuiCor` de la figure 11.3. Dans EPHORAT ce sommet est décrit à un niveau de granularité inférieur par trois instructions (`T_inter`, `T_C0` et `T_Ctau`). Après avoir spécifié le graphe d'algorithme, le graphe d'architecture et leurs caractéristiques, SynDEx effectue automatiquement la distribution et l'ordonnancement du graphe d'algorithme d'entrée en réseau sur l'architecture composée du Marañon (SIMD) et du mAgic-FPU, puis présente ces résultats sous la forme d'un diagramme de simulation temporelle (Cf. bas de la figure 11.3). Comme on le voit sur le graphe de l'algorithme, celui-ci ne présente pas de parallélisme potentiel. Après distribution/ordonnancement on constate que le mAgic-FPU doit attendre la fin de l'exécution de `Corr` avant de pouvoir exécuter `Cond`. Symétriquement, le Marañon doit attendre la fin de l'exécution

de pond pour pouvoir exécuter filtPuiCor . Par conséquent il est impossible d'exploiter le parallélisme disponible de l'architecture ce qui conduit à un temps d'exécution (216ms) supérieur à la contrainte temps réel qui était fixé ici à 170ms. Enfin, on constate que les communications entre le Marañon et le mAgic-FPU introduisent d'importants temps d'exécution et nécessitent des mémoires SAM de tailles importantes incompatibles avec celles disponibles dans Mefisto.

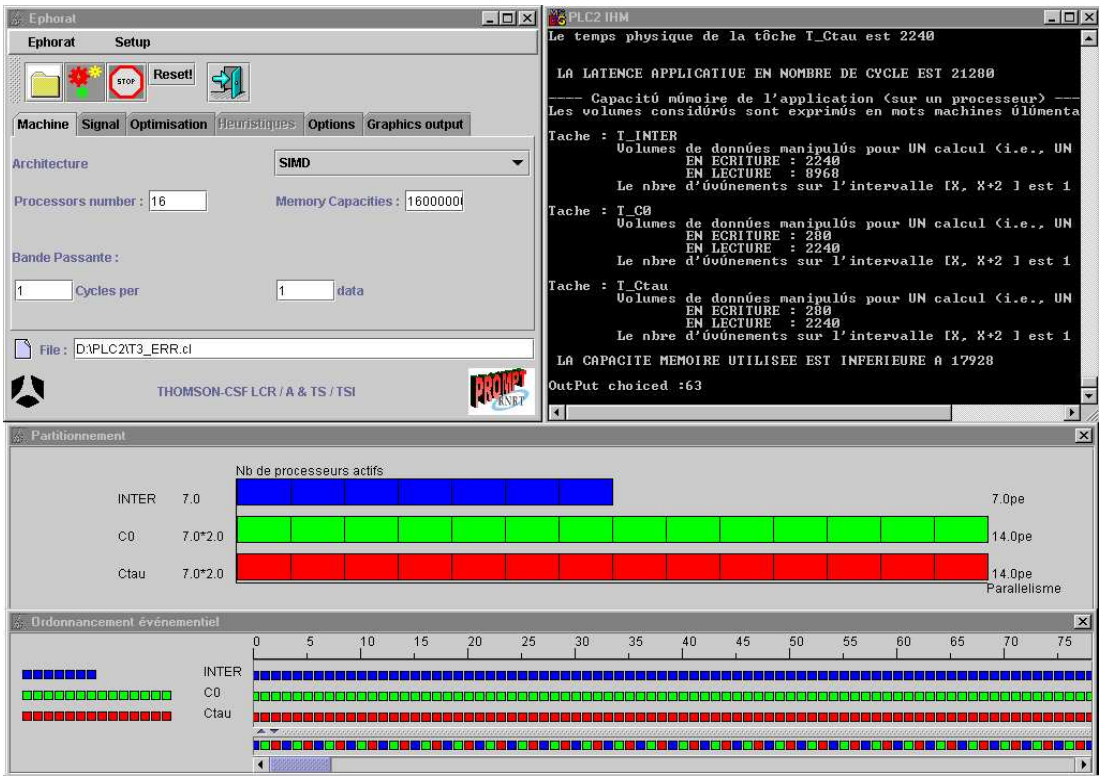


FIG. 11.4: *Logiciel EPHORAT*

Le couplage entre SynDEx et EPHORAT doit aider l'utilisateur à résoudre ces problèmes (respect de la contrainte temps réel par exploitation du parallélisme disponible, minimisation des communications et des mémoires) en lui permettant par exemple de modifier la spécification de son algorithme pour y introduire du parallélisme potentiel dont la granularité est compatible avec le parallélisme disponible de l'architecture (Cf. granularité 2.1.3, page 53). Dans notre exemple, comme toutes les opérations sont constituées de nids de boucles parfaitement imbriquées, il est possible de les découper pour créer des opérations plus petites (c'est à dire diminuer la granularité) qui pourront ainsi être exécutées en parallèle. Comme le montre le haut de la figure 11.5, nous avons découpé Corr en deux opérations, Corr1 et Corr2 , pond en pond1 et pond2 etc. Cette transformation introduit du parallélisme potentiel que SynDEx va maintenant pouvoir exploiter pour construire une implantation qui respecte la contrainte temps réel comme le montre la simulation temporelle du bas de la figure 11.5 (temps de réponse de 216ms dans le premier cas contre 167ms dans ce second cas). Les exécutions des opérations peuvent maintenant se recouvrir, la latence de l'application est donc plus faible, et la taille des mémoires SAM nécessaires entre le Marañon et le mAgic-FPU est aussi plus faible.

Cet exemple montre que le couplage des deux outils permet de programmer des applications complexes sur des SOC hétérogènes comprenant des processeurs SIMD, et permet de remettre en cause la structure

de l'algorithme afin d'exploiter le parallélisme disponible des SOC tout en respectant une contrainte temps réel.

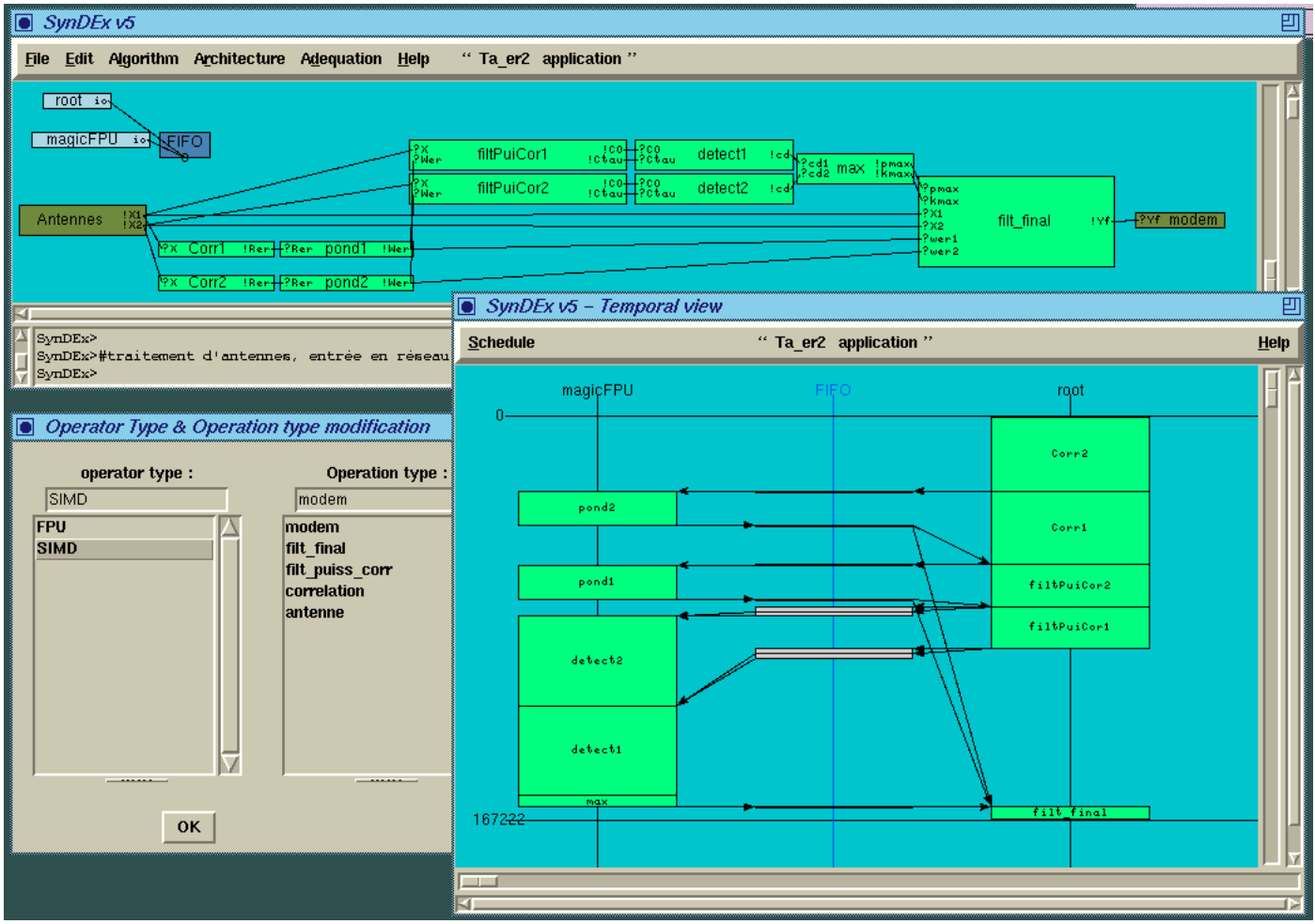


FIG. 11.5: *SynDEx : Implantation optimisée d'un algorithme de télécommunications*

11.2 Cycab, véhicule électrique public semi-automatique

11.2.1 Présentation

Le cycab[57] [56] (Cf. figure 11.6) est un petit véhicule électrique conçu et réalisé par l'INRIA Rocquencourt et l'INRIA Rhône-Alpes pour lequel différentes applications innovantes ont été réalisées (conduite manuelle à l'aide d'un joystick, suivi automatique d'un autre Cycab à l'aide d'une caméra embarquée, conduite automatique basées sur de la localisation par odométrie et de téléopération par radio).

Ce véhicule dispose d'une direction électrique et de quatre roues motrices directives. Chacun de ses cinq moteurs (un pour chaque roue et un pour la direction) est associé à des capteurs (vitesse, position) et un frein. Pour minimiser les câblages, la commande de chaque moteur à été distribuée sur un microcontrolleur associé aux capteurs et actionneurs nécessaires. Son architecture matérielle est composée de 5 processeurs (un Motorola 68332 par moteur) et un PC (basé sur un processeur 486DXII66) pour gérer une interface

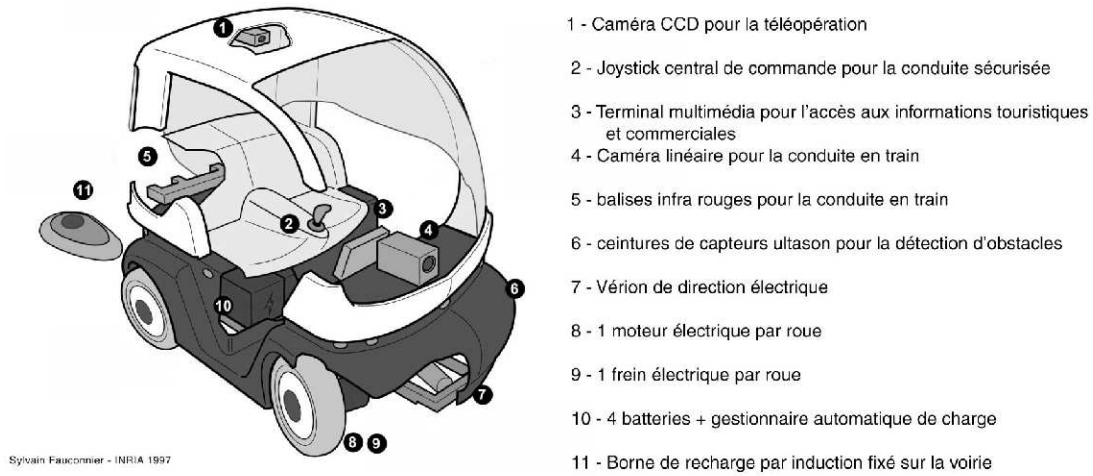


FIG. 11.6: *Le Cycab*

homme-machine interconnectés par un bus CAN (Cf. figure 11.7). Pour des applications complexes, une caméra vidéo associée à un processeur 80C96 a été connectée aux bus CAN.

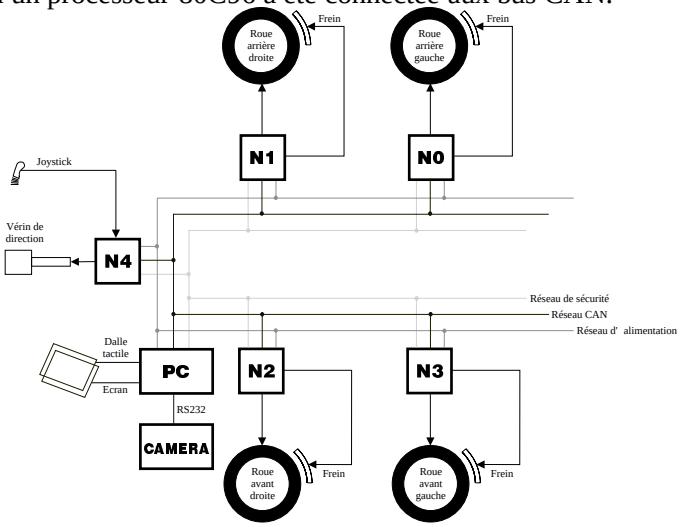


FIG. 11.7: *Architecture du Cycab (N0 à N4 : processeur 68332)*

11.2.2 Optimisations

La méthodologie AAA et le logiciel SynDEx ont été utilisés pour résoudre les problèmes que posent l'implantation d'algorithmes temps réel sur ce type d'architecture distribuée embarquée et hétérogène. Ainsi on a pu exploiter toute la bande passante du bus CAN (utilisée à seulement 60 pourcent avec les techniques de développement classique qui requièrent d'importantes marges de sécurité), ce qui à permis de transmettre des images sur le bus tout en effectuant le contrôle des moteurs. La figure 1.42 de la page 47 présente la modélisation de l'architecture du Cycab. La génération automatique d'exécutif taillé sur mesure conduit à

un exécutif utilisant moins de 20 kilo-octets de mémoire dans chaque processeur. Enfin, l'automatisation de la génération de code a permis dans la phase de mise au point, de développer rapidement des versions de l'application n'utilisant pas tous les processeurs quand ceux-ci n'étaient pas en état de fonctionner. Symétriquement, on a pu expérimenter plusieurs applications en conservant la même architecture matérielle.

Suite au succès de ce projet, une nouvelle version du Cycab est en cours de réalisation par l'industriel Robosoft. Elle est basée sur une architecture distribuée composée de quatre processeurs PowerPc de Motorola (MPC555), un pour chaque train, connectés à un PC Pentium sous RTAI/Linux par deux bus CAN afin d'étudier la tolérance aux pannes par redondance matérielle.

Chapitre 12

Évaluation quantitative de la méthodologie AAA

Sommaire

12.1 Introduction

Il existe deux manières d'évaluer quantitativement une méthodologie, relativement à d'autres méthodologie du même type, ou bien de façon absolue.

Par ailleurs, il faut définir les critères utilisés pour cette évaluation. Il nous semble ici intéressant d'évaluer d'une part le temps de développement d'une application, et d'autre part ses performances, c'est à dire le nombre de ressources (mémoires programme et mémoires de données communiquées, nombre de processeurs) utilisées pour respecter les contraintes temps réel.

L'évaluation relative sur les temps de développement s'est avérée impossible à réaliser car il n'existe pas d'outil équivalent, prenant en compte toute la chaîne de développement d'une application en partant de la spécification haut niveau de l'algorithme et de l'architecture, jusqu'à la génération de l'exécutif temps réel distribué optimisé pour chaque processeur de l'architecture, comme cela est possible avec la méthodologie Adéquation Algorithme Architecture et le logiciel SynDEx qui la supporte.

En revanche nous pouvons donner une évaluation absolue des temps de développement et des performances obtenus pour quelques applications.

12.2 Temps de développement pour l'application Cycab

Le développement de l'application de conduite manuelle sur le nouveau Cycab a été réalisé par une équipe ne connaissant ni la méthodologie AAA et le logiciel SynDEx ni le processeur MPC555, le PC sous

RTAI/Linux et le bus CAN qui forment l'architecture distribuée hétérogène du Cycab. Pour présenter les résultats, nous avons découpé le développement de cette application en trois étapes :

- spécification, optimisation, génération des exécutifs avec SynDEx,
- réalisation des macros systèmes pour chaque composant de l'architecture : processeurs MPC555, processeur Pentium sous RTAI/Linux et communications par bus CAN du côté MPC555 et du côté RTAI/Linux,
- réalisation des macros applicatives spécifiques au Cycab.

12.2.1 Spécification, optimisation et génération d'exécutifs SynDEx

Quelques jours ont été nécessaires pour spécifier l'application avec SynDEx. Des variantes de cette application ont été réalisées ensuite en quelques heures, parfois quelques minutes selon leur complexité. Après un temps d'adaptation au logiciel qui est facile à maîtriser, les temps de spécification ont encore diminués. L'optimisation de la distribution et de l'ordonnancement puis la génération des exécutifs distribués temps réel optimisés ne prend que quelques minutes. Le générateur d'exécutif utilise les macros systèmes et les macros développées par ailleurs. Enfin, l'exécution elle même s'effectue simplement en lancant la commande "make" qui utilise un makefile généré automatiquement par SynDEx.

12.2.2 Réalisation des macros systèmes

12.2.2.1 Processeur MPC555

Le portage de l'exécutif pour le processeur MPC555 a requis trois mois, sans aucune connaissance préalable du fonctionnement de ce processeur. Soulignons que pour minimiser la taille des exécutables et les durées d'exécutions, l'exécutif MPC555 a entièrement été écrit en assembleur, ce qui constitue une difficulté supplémentaire.

12.2.2.2 Communications CAN par MPC555

Le portage de la partie de l'exécutif qui gère les communications sur le bus CAN coté MPC555 à duré trois mois. La difficulté était due essentiellement à la difficulté de gérer les interruptions du MPC555 qui sont particulièrement complexes à maîtriser.

12.2.2.3 Processeur PC Pentium sous RTAI/Linux

Comme la taille des ressources mémoire disponibles sur le PC est moins critique que celles disponibles dans les MPC555, cet exécutif à été écrit en langage C plutôt qu'assembleur. Ceci, ajouté à une meilleure connaissance des principes de SynDEx a permis de réduire sensiblement la durée des développements qui est passée à seulement trois mois pour l'exécutif RTAI/Linux avec support des communications par bus CAN.

12.2.3 Réalisation des macros applicatives

Le temps de développement des macros applicatives est le plus variable puisqu'il correspond au temps de codage des opérations spécifiques à l'algorithme de l'application. Dans le cas de l'application de conduite manuelle du Cycab, il a été nécessaire d'écrire un grand nombre de macros d'entrée-sortie (capteur-actionneur)

pour gérer l'acquisition de la position du joystick, la commande des moteurs en PWM (modulation en largeur d'impulsions), l'acquisition de la vitesse de rotation de chaque moteur, du courant circulant dans les induits de chaque moteur, de l'angle de direction des essieux avant et arrière, ainsi que de l'asservissement des moteurs de traction et de direction. Ce grand nombre de macros s'est donc traduit par un temps de développement relativement élevé de quatre mois. Soulignons néanmoins que toutes ces macros applicatives, comme les macros systèmes développées précédemment, sont directement réutilisables pour les futurs applications Cycab.

12.3 Performances

12.3.1 Application Cycab

L'application Cycab repose sur des asservissements effectués à une cadence d'une milliseconde. La taille des données communiquées sur le bus CAN est de 32 octets. La taille des exécutifs générés et chargés automatiquement dans chacun des quatres MPC555 est de l'ordre de 3 Koctets, celui du PC sous RTAI/Linux est de 6 Koctets. Enfin, le surcoût d'exécution pour un changement de contexte entre une séquence de calcul d'un opérateur et d'une séquence de communication d'un communicateur est inférieur à 100 cycles-instructions dans l'exécutif pour processeur MPC555. Ceci inclut les opérations de synchronisations, la programmation du DMA, l'interruption de fin de transfert et la libération des sémaphores indiquant la fin de la communication.

12.3.2 Application de traitement d'images

Pour une application temps réel (cadence vidéo) de segmentation d'image par les contours sur une architecture composée de quatre processeurs DSP TMS320C40, la taille des exécutifs générés et chargés automatiquement dans chacun des TMS320C40 varie de 5 à 7 Koctets. Le surcoût d'exécution pour un changement de contexte entre une séquence de calcul d'un opérateur et d'une séquence de communication d'un communicateur est de 56 cycles-instructions dans l'exécutif pour processeur DSP TMS320C40.

Conclusion

La méthodologie Adéquation Algorithme Architecture permet d'aider le concepteur à implanter de façon optimisée les algorithmes complexes de contrôle/commande et de traitement du signal et des images sur des architectures distribuées hétérogènes embarquées, en respectant des contraintes temps réel. Elle repose sur un formalisme de graphes pour décrire l'architecture matérielle l'algorithme de l'application, ainsi que toutes les transformations (incluant les optimisations) qui permettent la génération automatique d'exécutifs. Nous citons ci-dessous les principaux apports de cette thèse.

Spécification, implantation et optimisation

Afin d'élargir le champs d'application de la méthodologie, d'améliorer la qualité de ces optimisations, de permettre de nouvelles optimisations, et de permettre la génération automatique d'exécutifs, les modèles de la méthodologie AAA ont été enrichi et des modifications et améliorations des heuristiques de distribution ordonnancement ont été apportées.

Le modèle d'architecture est maintenant plus précis que celui utilisé précédemment dans la méthodologie AAA, il est bien adapté à l'optimisation de la distribution et de l'ordonnancement hors ligne, ainsi qu'à la génération automatique d'exécutifs. Il permet en effet de décrire précisément les mécanismes d'arbitrage à différents niveaux d'une machine, augmentant ainsi la qualité des prédictions temporelles et par conséquence les possibilités d'optimisation. L'introduction des mémoires RAM dans le modèle permet maintenant de faire de l'optimisation sur la distribution des données par réutilisation de la mémoire. La diversité des mémoires traitées (SAM, RAM, point à point, multipoint, avec ou sans support du broadcast) permet de supporter maintenant un plus grand nombre d'architectures matérielles, tout en offrant la possibilité d'effectuer des optimisations plus fines et adaptées à chaque type de média, ce qui augmente encore la qualité de l'optimisation.

Le modèle d'algorithme a été enrichi de façon à permettre la génération automatique d'exécutifs.

Le modèle d'implantation a été enrichi en corrélation avec les nouveaux modèles d'architectures et d'algorithmes, d'une part de façon à faire le plus d'optimisations possibles pour exploiter le parallélisme de communication offert par l'architecture et réutiliser des communications dans le cas de données diffusées, et d'autre part pour utiliser au mieux les différentes mémoires.

L'heuristique a été largement enrichie pour prendre en compte les nouvelles informations offertes par la modélisation plus fine de l'architecture. Ainsi, les mémoires et leurs caractéristiques sont maintenant prises en compte dans l'heuristique de distribution/ordonnancement. Une phase supplémentaire d'optimisation, basée sur des techniques de ré-allocation statique de la mémoire a aussi été ajoutée pour minimiser d'avantage la taille des mémoires.

Génération automatique d'exécutifs distribuées

Le processus complet de génération d'exécutifs à été modélisé et validé par transformation de graphes, en présentant d'abord chaque règle de construction de l'exécutif et en faisant la preuve de la conservation des

propriétés du graphe d'algorithme à chaque étape. La génération automatique de l'exécutif dans le contexte des applications temps réel distribuées embarquées hétérogènes est bien adaptée au prototypage rapide. Cela a été démontré par la qualité de l'exécutif, sa généricité, et de sa compacité. La structure de l'exécutif a été étudiée en détail, ainsi que tous les principes qui permettent de le rendre générique.

Développement logiciel

Enfin, des développements logiciel ont été réalisés et validés dans le cadre de deux applications réalistes issues du monde industriel. Le résultat est une nouvelle version (la V5) du logiciel de CAO niveau système SynDEx. Il a été conçu en respectant les critères de génie logiciel (spécification détaillée, documentation) afin de l'ouvrir au maximum et de permettre son évolution. Ainsi, certains chercheurs ont déjà pu expérimenter de nouvelles heuristiques et ajouter des fonctionnalités spécifiques à certains domaines. Par exemple, il a été introduit la possibilité de spécifier des automates pour décrire la partie contrôle des algorithmes [56].

Voici maintenant quelques perspectives en vue de prolonger ce travail. Dans cette thèse nous nous sommes restreint à l'ordonnancement hors ligne non préemptif des calculs et des communications avec une seule contrainte temporelle de latence. Dans le but d'élargir le champ d'application de notre méthodologie, il faudrait introduire la possibilité de spécifier des applications devant respecter plusieurs contraintes temporelles. Pour cela il sera nécessaire d'introduire de la préemption dans les ordonnancements hors-ligne, en essayant de minimiser son effet, car son coût est loin d'être négligeable.

Nous avons commencé à étudier les aspects concernant la tolérance aux pannes dans le cadre du projet TOLERE [38]. Plus précisement nous travaillons sur des heuristiques qui cherchent à utiliser les redondances matérielles de l'architecture pour construire des ordonnancements tolérants la panne d'un certain nombre de processeurs et/ou de média de communication.

Pour élargir d'avantage le champ des architectures il reste à enrichir le modèle d'architecture avec des circuits spécifiques non programmables, parfois reconfigurables, tels que des ASIC ou des FPGA, qui sont de plus en plus utilisés dans les architectures temps réels embarquées. On cherchera à unifier les aspects programmables et non programmables des modèles, de manière à poser formellement (le plus objectivement possible) le problème de la conception conjointe logiciel/matériel.

Bibliographie

[1] Behrooz A. Shirazi, A. Hurson, and Krishna M. Kavi. *Scheduling and load balancing in parallel and distributed system*. IEEE Computer Society Press, 1995.

[2] R. Airiau, J.-M. Berge, V. Olive, and J. Rouillard. *VHDL, du langage à la modelisation*. Presses polytechniques et universitaires romandes, 1990.

[3] C. Ancourt, M. Barreteau, B. Dion, T. Grandpierre, F. Irigoin, J. Jourdan, P. Kajfasz, C. Lavarenne, and Y. Sorel. Prompt : Placement rapide optimisé sur machines parallèles pour applications de télécommunications. In *AAA2000, Actes du 5ème Worshop sur l'Adéquation Algorithme Architecture*, Paris, janvier 2000.

[4] C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jeannet, J. Jourdan, and J. Mattioli. Automatic data mapping of signal processing applications. In *IEEE International Conference on Application Specific Systems, Architectures and Processors*, pages 350–362, Zurich, Switzerland, July 1997.

[5] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real time systems. *IEEE Design and Test of Computers*, pages 71–82, January-March 1998.

[6] M. Barreteau, P. Bonnot, T. Grandpierre, P. Kajfasz, C. Lavarenne, J. Mattioli, and Y. Sorel. Prompt : A mapping environment for telecom applications on "system on a chip". In *CASE2000, International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, Cal. US, november 2000.

[7] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proc. of the IEEE*, volume 79(9), pages 1270–1282, 1991.

[8] S. Bhattacharya, K.P. Murthy, and E.A. Lee. *Software synthesis from dataflow graphs*. Kluwer Academic, 1996.

[9] J. Blazewicz, K. Ecker, and D. Trystram B. Plateau. *Handbook on parallel and distributed processing*. Springer, 2000.

[10] N.S. Bowen, C.N. Nikolaou, 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.

[11] G.W. Brams. *Réseaux de Pétri : Modélisation et application*, volume 2. Masson, 1983.

[12] G.W. Brams. *Réseaux de Pétri : Théorie et pratique*, volume 1. Masson, 1983.

[13] X. Briffault and G. Sabah. *SMALLTALK, programmation orientee objet et developpement d'applications*. Eyrolles, 1996.

[14] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 8(C-35):677–691, August 1986.

[15] J.T. Buck, S. Ha, E.A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Int. Journal of Computer Simulation, special issue on "Simulation Software Development*, volume 4, pages 155–182, April 1994.

[16] J. Carlier and P. Chretienne. *Problèmes d'ordonnancement*. Masson, 1988.

[17] T.L. Casavant and J.G. Kuhl. A taxonony of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Eng.*, 14, No. 2:141–154, Feb. 1988.

[18] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective CAML*. O'Reilly, 2000.

[19] B. Charron-Bost. *Mesures de la concurrence et du parallélisme des calculs répartis*. Thèse d'informatique, Université Paris VII, Septembre 1989.

[20] G. Cornell and C.S. Horstmann. *Core JAVA*. Sunsoft Press, 1997.

[21] M. Cosnard and A. Ferreira. The real power of loosely coupled parallel architectures. In *Parallel Processing Letters*, pages 103–112, 1991.

[22] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles*. InterEditions, PARIS, 1993.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp : Toward a realistic model of parallel computation. In *Proceeding of 4-th ACM SIGPLAN, Symposium on Principles and Practises of Parallel Programming*, pages 1–12, 1993.

[24] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy II: Heterogeneous concurrent modeling and design in java. Technical Report Technical Report UCB/ERL No. M99/40, University of California, Berkeley, CA 94720, July 1999.

[25] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel ceometric algorithms for coarse grained multicomputer. In *ACM 9th Symposium on Computational Geometry*, pages 298–307, 1993.

[26] G. Delamarre. *Dictionnaire des réseaux, Télématique, RVA EDI*. Collection Transpac, 1980.

[27] 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.

[28] A.F. Dias. Contribution à l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-fpga à l'aide d'une méthodologie d'Adéquation Algorithme Architecture. Master's thesis, Université de Paris-Sud, Orsay, France, Juillet 2000.

[29] R. Djenidi, C. Lavarenne, R. Nikoukhah, Y. Sorel, and S. Steer. From hybrid system simulation to real-time implementation. In *ESS'99. SCS*, Erlangen, Germany, 1999.

[30] J.R. Ellis. *BULLDOG: a Compiler for VLIW Architectures*. MIT Press, Cambridge, 1986.

[31] F. Ennesser, C. Lavarenne, and Y. Sorel. Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs syndex. Technical Report 1769, INRIA, 1992.

[32] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test of Computers*, pages 45–53, April-June 1998.

[33] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Septembre 1972.

[34] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10-th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[35] V. Fresse, M. Assouil, and O. Deforges. Rapid prototyping for mixed architectures. In *25th IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP)*, Istambul, Turkey, 05-09 june 2000.

[36] Garey and Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[37] A. Geist, A. Beguelin, and J. Dongarra. *PVM - parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, 1994.

[38] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. Rapport de Recherche 4006, INRIA, Septembre 2000.

[39] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, 1979.

[40] T. Grandpierre. *Spécification en langage orienté objet pour le portage du cœur de SynDEx*, Université d'Orsay edition, Août 1996. Rapport de DEA.

[41] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de Recherche 3476, INRIA, Août 1998.

[42] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design*, pages 74–78, Rome, Italy, May 3-5 1999. IEEE Computer Society, ACM SIGSOFT, IFIP.

[43] C. Guettier. *Optimisation global du placement d'applications de traitement du signal sur architectures parallèles utilisant la programmation logique avec contraintes*. PhD thesis, École des mines de Paris, Décembre 1997.

[44] N. Halbwachs. *The declarative code DC, version 1.2a*. Vérimag, Grenoble, France, October 1995. unpublished report, http://www.inrialpes.fr/bip/people/girault/Documentations/Dc1/index.html.

[45] M. Harrison and M. McLennan. *Effective Tcl/Tk programming, writing better programs with Tcl and Tk*. Addison-Wesley, 1998.

[46] R.H. Hartenstein. *Fundamentals of structured hardware design: a design language approach at register transfer level*. North-Holland, 1977.

[47] S. Heath. *Microprocessor architectures RISC, CISC and DSP*. Newnes-Butterworths, 2nd edition, 1995.

[48] J.L. Hennessy and D.A. Patterson. *Organisation et conception des ordinateurs, l'interface materiel-logiciel*. International Thomson Publishing France, 1994.

[49] J.L. Hennessy and D.A. Patterson. *Architecture des ordinateurs, approche quantitative*. International Thomson Publishing France, 2nd edition, 1996.

[50] HICSS 1995. *Models of Parallel Computation: A Survey and Synthesis*, volume 2, 1998.

[51] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[52] IEEE. VHDL, manuel de référence. Technical report, IEEE-1076.

[53] J.L. Jacquemin. *Informatique parallele et systemes multiprocesseurs*. Hermes, 1993.

[54] E.E. Johnson. Completing an mimd multiprocessor taxonomy. *Computer Architecture News*, 16(3):44–47, Juin 1988.

[55] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Proc. of Int. Conference on Parallel Processing*, volume 3, pages 1–8, August 1988.

[56] R. Kocik. *Sur l'optimisation des systèmes distribués temps réel embarqués : application au prototypage rapide d'un véhicule électrique autonome*. PhD thesis, Université de Rouen, Mars 2000.

[57] R. Kocik and Y. Sorel. A methodology to design and prototype optimized embedded robotic systems. In *Proc. of the Computational Engineering in Systems Applications CESA'98, Tunisia*, April 1998.

[58] Y. Kopidakis, M. Lamari, and V. Zissimopoulos. On the task assignment problem: Two new efficient heuristic algorithms. *J. of Parallel and Distributed Computing*, 42:21–29, 1997.

[59] Y. Kwok, I. Ahmad, M.Y. Wu, and W. Shu. A graphical tool for automatic parallelization and scheduling of programs on multiprocessors. In Hank Dietz, editor, *International conference on parallel processing 26 ; 1997 ; Bloomingdale, IL US*, pages 294–301. IEEE computer society press, 1997.

[60] Y. K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[61] 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 Int. Conference on Parallel Processing*, volume 2, pages 150–157, August 1996.

[62] Lockheed Martin Advanced Technology Laboratories. Gedae technical paper. Technical report, Lockheed Martin Advanced Technology Laboratories., 1998.

[63] I. Guérin Lassous. *Algorithmes parallèles de traitement de graphe : une approche basée sur l'analyse expérimentale*. PhD thesis, LIAFA,Université de Paris VII, 1999.

[64] C. Lavarenne and Y. Sorel. Specification, performance optimization and executive generation for real-time embedded multiprocessor applications with syndex. In *Proc. of Real-Time Embedded Processing for Space Applications*. CNES International Symposium, 1992.

[65] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Revue Traitement du Signal*, 14(6), 1997.

[66] C. Lavarenne and Y. Sorel. Performance optimization of multiprocessor, real-time applications by graph transformations. In *Proc. of PARCO93 conference*, September 1993.

[67] B. Lee, A.R. Hurson, and T.-Y. Feng. A vertically layered allocation scheme for data flow systems. *Parallel and Distributed Computing*, 11(3):175–187, 1991.

[68] E.A. Lee and J.C. Bier. Architectures for statically scheduled dataflow. *J. of Parallel and Distributed Computing*, 10:333–348, 1990.

[69] P. Leguernic, T. Gautier, M. Leborgne, and C. Lemaire. Programming real-time applications with SIGNAL. *Proc. IEEE*, 79(9):1321–1336, September 1991.

[70] J.K. Lenstra and A.H.G. Rinnoy Kan. Complexity of scheduling under precedence constraints. In *Operations Research*, pages 22–35. Janvier 1978.

[71] R. Leupers. *Retargetable code generation for digital signal processing*. Kluwer Academic Publ., 1997.

[72] D. Lewine. *POSIX Programmer's Guide*. O'Reilly, 1991.

[73] M. Loukides and A. Oram. *Programmer avec les outils GNU*. O'Reilly, 1997.

[74] Z. Lui and C. Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. *Proc. of PARLE'93, 5th Int. PARLE conference, Munich, Germany, June 14-17*, pages 452–463, Nov. 1993.

[75] M. J. Murdocca and V. P. Heuring. *Principles of computer architecture*. Prentice Hall, 2000.

[76] 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.

[77] R. Nikoukhah and S. Steer. SCICOS, a dynamic system builder and simulator user's guide - version 1.0. Technical Report RT 207, INRIA, 1997.

[78] A. Oram and S. Talbott. *Managing projects with MAKE*. O'Reilly and Associates, 1995.

[79] J. K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.

[80] D. Paret. *Le bus I2C*. Dunod, 1994.

[81] Parsytec. http://www.parsytec.de/top/top.htm.

[82] D.A. Patterson and C.H. Sequin. A VLSI RISC. *IEEE Computer Magazine*, 15(9):8–21, 1982.

[83] L. Phelippeau-Gelineau. *Etude de problèmes d'ordonnancement multiprocesseur avec communication par diffusion*. PhD thesis, Université Paris VI, 1996.

[84] G. Radin. The 801 minicomputer. In ACM SIGARCH-10.2 SIGPLAN-17.4, editor, *Symposium on Architectural Support for Programming Languages ans Operating Systems (ASPLOS-I)*, pages 39–47, 1982.

[85] RASSP, editor. *2nd ANNUAL RASSP CONFERENCE PAPERS*, 1995. http://rassp.scra.org/.

[86] R. Rockafellar and Tyrrell. *Convex analysis*. Princeton University Press, 1972.

[87] E. Rougerie. *Etude et réalisation d'un environnement logiciel pour machine massivement parallèle Multi-SIMD*. PhD thesis, Université de Paris XI Orsay, 1993.

[88] J. Rumbaugh, M. Blaha, F. Eddy, and al. *OMT, modelisation et conception orientees objet*. Masson, 1995.

[89] J. Rumeur. *Communications dans les réseaux de processeurs*. Etudes et recherches en informatique, Masson edition, 1994.

[90] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[91] L. Schäfers and C. Scheidler. Trapper: A graphical programming environment for embedded MIMD computers. In S.C. Hilton M.R. Jane R. Grebe, J. Hektor and P.H. Welch, editors, *Transputer Applications and Systems'93*, pages 1023–1034. Proceedings of the 1993 World Transputer Congress, IOS Press, 1993.

[92] M. Snir, S. Otto, S. Huss-Lederman, and D. Walker. *MPI: the complete reference*. MIT Press, 1996.

[93] Y. Sorel. Massively parallel computing systems with real time constraints, the algorithm architecture adequation methodology. In *Proc. of the Massively Parallel Computing Systems*, May 1994.

[94] Y. Sorel. Real-time embedded image processing applications using the A^3 methodology. In *Proc. of the IEEE Int. Conference on Image Processing*, November 1996.

[95] M. Sorine and Y. Sorel. Rapport annuel. Technical report, INRIA, 1995.

[96] J. Sérot. Un compilateur caml : Syndex pour les applications de traitement du signal distribuées. In *11^{eme} Journées Francaises des Langages Applicatifs*, 1er February 2000.

[97] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.

[98] Texas Instruments. *TMS320C80 User's guide*, 1995.

[99] A. Tiskin. The bulk-synchronous parallel random access machine. In Springer, editor, *Europar'96*, pages 327–338, 1996.

[100] J. Tsay. A code generation framework for ptolemy II. Technical Report ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, 2000.

[101] L. Valiant. A bridging model for parallel computation. In *Communications of the ACM*, volume 33 (8), pages 103–111. 1990.

[102] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. PhD thesis, Université Paris XIII, Juillet 1999.

[103] T. Yang. *Scheduling and code generation for parallel architectures*. PhD thesis, University of New Jersey, New Brunswick, New Jersey, May 1993.

[104] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing Journal*, 19:1321–1344, 1993.

[105] 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.

[106] H. Zeltwanger. An inside look at the fundamentals of CAN. *Control Engineering*, pages 51–56, January 1995.

[107] A.Y Zomaya. *Parallel and distributed computing handbook*. McGraw-Hill, 1996.

[108] 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.