INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Utilisation de SynDEx pour le traitement d'images temps-réel

Caroline AIGLON, Christophe LAVARENNE, Yves SOREL, Annie VICARD

N° 2968

Septembre 1996

THEME 4

ISSN 0249-6399

Utilisation de SynDEx pour le traitement d'images temps-réel

Caroline AIGLON, Christophe LAVARENNE, Yves SOREL, Annie VICARD

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet SOSSO

Rapport de recherche n ° 2968 — Septembre 1996 — 79 pages

Résumé : ce rapport montre comment le logiciel SynDEx apporte de l'aide lors de l'implantation d'algorithmes de traitement d'images devant respecter des contraintes temps réel. La méthodologie A^3 (Adéquation Algorithme Architecture) sur laquelle SynDEx est fondé, grâce à un formalisme unifié de transformations sur des graphes, permet une approche globale du problème d'implantation d'algorithmes sous contrainte temps réel. Ceci conduit, d'une part à de l'aide pour la parallélisation, la distribution et l'ordonnancement de l'algorithme sur l'architecture et pour le dimensionnement des ressources matérielles nécessaires, et d'autre part à la génération automatique d'exécutifs distribués principalement statiques, donc à faible surcoût. Ces derniers gèrent entre autre, un système de communications interprocesseurs efficace avec prévention des interblocages. Pour illustrer les avantages de cette approche, nous avons choisi d'implanter un algorithme de détection de contours, représentatif du traitement d'images bas et moyen niveau. Nous avons montré qu'à partir d'une version fonctionnant en mono-processeur de cet algorithme, il est facile d'en faire avec SynDEx une version multi-processeur, qui possède le même comportement en termes des valeurs produites et qui approche au mieux le temps réel avec les quatre processeurs de traitement du signal TMS320C40 dont nous disposions. Cette implantation multi-processeur est mise en œuvre sans avoir à faire de mise au point multi-processeur. La démarche présentée réduit de manière très importante les temps de développement des applications de traitement d'images temps réel.

Mots-clé : traitement d'images, traitement du signal, commande de processus, langage synchrone, SIGNAL, vérification de programme, architecture parallèle et distribuée, multi-composant, multi-processeur, contraintes temps réel, distribution et ordonnancement, optimisation, SynDEx.

(Abstract: pto)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Using SynDEx for real-time image processing

Abstract: this report shows how the SynDEx software may aid to the implementation of image processing algorithms, in real time. SynDEx is based on the Algorithm Architecture Adequation methodology ("adequation" is a french word meaning an efficient mapping) which leads to a global approach for the implementation under real-time constraints, formalized in terms of graphs transformations. SynDEx helps to parallelize, to distribute and schedule the algorithm on the architecture, and to minimize the hardware resources. It automatically generates distributed executives, mainly static for low over-head, which manage a dead-lock free inter-processor communication system. We chose an edge detection algorithm, representative in signal processing, to stress the benefits brought by the approach. That is, from a uniprocessor implementation of this algorithm, it was very easy with SynDEx to derive a multiprocessor implementation which behaviour was identical and approaching as close as possible real-time, while fully using the power provided by the four TMS320C40 DSPs at our disposal. This implementation does not require any debugging. The presented approach dramatically reduces the development cycle of such image processing applications running in real-time.

Key-words: image processing, signal processing, process control, synchronous language, SIGNAL, program verification, parallel and distributed architecture, multi-processor, real-time constraints, distribution and scheduling, optimization, SynDEx.

1 Introduction

Un des objectifs du projet SOSSO concerne l'étude des problèmes posés par l'implantation sous contraintes temps-réel, d'algorithmes de contrôle commande et de traitement du signal et des images s'exécutant sur des architectures *multi-composants* embarquées. Les applications concernées sont des systèmes réactifs temps réel nécessitant des architectures matérielles complexes qui font intervenir du parallélisme. Ces études ont donné lieu à une méthodologie appelée Adéquation Algorithme Architecture (A^3) [1], mise en œuvre dans le logiciel d'aide à l'implantation SynDEx (acronyme pour EXécutif Distribué SYNchrone) [1].

On montre dans ce rapport comment la méthodologie A^3 ainsi que le logiciel SynDEx, peuvent aider à l'implantation d'algorithmes de traitement d'images devant s'exécuter en temps réel, rendue particulièrement difficile à cause du grand nombre de données mises en jeu et de l'importante puissance de calcul nécessaire pour respecter les contraintes temps réel. Afin de valider la démarche, on a choisi d'implanter un algorithme de "détection de contours" sur une architecture multi-DSP. Cet algorithme, dit de bas niveau, est classiquement utilisé dans les chaînes complètes de traitement d'images. Nous l'avons choisi car il soulève les problèmes standards que l'on rencontre en traitement d'images temps réel.

Le rapport est organisé comme suit. On commence dans la suite de cette introduction par définir les termes importants. On donne ensuite les principes de la méthodologie A^3 basée sur des modèles de graphe. Ces modèles de graphe sont utilisés pour spécifier aussi bien les algorithmes que les architectures matérielles. On montre ensuite comment formaliser l'implantation d'un algorithme sur une architecture en termes de transformations de graphes. On présente le composant (DSP Digital Signal Processor et lien de communication) permettant de construire l'architecture cible sur laquelle l'algorithme sera implanté. On présente comment spécifier et vérifier l'algorithme de détection de contours avec le langage SIGNAL, indépendamment des contraintes matérielles et temps réel. Enfin, on montre comment le logiciel SynDEx est utilisé pour aider à l'implantation de l'algorithme spécifié avec SIGNAL, en prenant en compte le matériel, les contraintes temps réel (latence) et en minimisant les composants matériel. On montre aussi comment SynDEx peut aider à choisir la granularité du parallélisme lors de la spécification de l'algorithme. Avant de conclure, on donne quelques éléments concernant les performances de cette implantation.

Système réactif temps réel

Un système **réactif** reçoit des stimuli en entrée, i.e. des événements venant de l'environnement, effectue des opérations et réagit en produisant des sorties i.e. des événements utilisables par l'environnement. Un système réactif est dit **temps réel** s'il réagit en respectant les contraintes de temps liées à l'application traitée.

Ces contraintes sont de deux types :

- **la latence** : intervalle de temps entre la réception de la donnée engendrée par un stimulus et l'émission de la donnée engendrée par une réaction à l'issue du traitement
- **la cadence** : intervalle de temps qui sépare la réception, par le système, de deux stimuli consécutifs

Remarque

La notion de temps réel ne peut donc pas être définie indépendamment du contexte dans lequel on se trouve. Une application temps réel dans le domaine de la vision robotique impose une latence de l'ordre de la milliseconde avec une cadence du même ordre de grandeur. En revanche, dans le domaine de la météorologie par exemple, le temps de réponse admis est de l'ordre de l'heure ou de la journée.

Parallélisme potentiel

Lorsqu'un algorithme est spécifié à l'aide d'un langage de programmation impératif (purement séquentiel), un ordre total est défini sur l'exécution de toutes les opérations à réaliser alors que certaines de ces opérations, n'étant *pas* en relation de dépendances de données (les calculs de l'une ne dépendent *pas* des résultats des calculs de l'autre), pourraient être exécutées en parallèle. Il est préférable d'utiliser un langage déclaratif décrivant un ordre partiel sur les opérations. Dans ce cas, on n'impose un ordre d'exécution sur deux opérations, que lorsqu'elles sont en relation de dépendances de données. Cet ordre partiel définit le parallélisme potentiel de l'algorithme. Les opérations qui ne sont pas en relation pourront être affectées à des ressources différentes et donc, être éventuellement effectuées en parallèle, si le parallélisme effectif de l'architecture le permet. Dans le cas contraire il faudra imposer un ordre d'exécution compatible avec l'ordre partiel de l'algorithme. Il faut noter que si l'algorithme est spécifié de manière impérative, il sera nécessaire d'effectuer une analyse de dépendance pour l'exécuter sur une architecture parallèle. La notion d'algorithme utilisée ici est une extension de la notion habituelle telle que définie par Turing par exemple. D'une part on considère qu'un algorithme est associé à un ordre partiel plutôt que total, et d'autre part la réunion d'un ensemble d'algorithmes est aussi appelée algorithme.

Traitement d'images bas niveau

Une chaîne de traitements d'images, fait intervenir classiquement trois niveaux d'algorithmes. Le niveau bas concerne les prétraitements (filtrage, gradient), ces traitements se font sur les données les plus fines (pixel). Le moyen niveau concerne l'extraction des informations significatives (contours fermés d'objets, régions), ces traitements se font sur des

données réduites (liste). Enfin le haut niveau concerne l'interprétation des informations extraites précédemment, pour prendre des décisions. A ce niveau on effectue généralement des traitements symboliques sur des données très agglomérées (comparaison de caractéristiques d'objets par rapport à une base de connaissances). Globalement on peut dire que plus on monte dans les niveaux, plus le traitement devient irrégulier et plus le nombre de données à traiter diminue.

L'algorithme de détection de contours considéré ici, concerne principalement le bas niveau. On effectue trois traitements en séquence. Après un filtrage passe bas destiné à supprimer le bruit, on recherche les maxima locaux de la norme du gradient en chacun des pixels, puis on réalise un affinage de ces maxima afin de mettre en évidence les contours. Ces traitements se font au niveau pixel. L'image de départ est traitée pixel par pixel donnant après chaque traitement une nouvelle image modifiée. La dernière image ne contient que les contours significatifs. Lors de la spécification et de l'implantation, on considèrera bien sûr cette chaîne de traitements comme un algorithme unique. Ces traitements effectués sont réguliers, c'est la même opération qui est appliquée à des données différentes (ici des pixels différents). Cela revient à spécifier l'algorithme directement en faisant apparaître son parallélisme potentiel en mode "data parallélisme". Évidemment le problème de la granulation du parallélisme se pose aussitôt car le grain pixel n'est pas celui le mieux adapté à une implantation efficace. Il faut remarquer que la notion de grain dépend à la fois des données traitées et de l'opération effectuée sur ces données. Nous verrons comment la méthodologie et le logiciel associé peuvent aider à prendre en compte les caractéristiques du matériel que l'on a à sa disposition afin d'exploiter le parallélisme potentiel de l'algorithme en fonction du parallélisme effectif de l'architecture. Avec cet exemple, nous n'avons pas chercher à réaliser une chaîne complète de segmentation d'images fonctionnant en temps réel sur une architecture multi-DSP mais plutôt à montrer comment la méthodologie A^3 ainsi que le logiciel SynDEx permettent d'aborder ces problèmes complexes.

Architecture multi-composant

Bien que les architectures mono-processeur (basées sur un processeur RISC d'une station de travail par exemple) soient de plus en plus performantes, elles ne peuvent répondre à l'accroissement de complexité constante de certaines applications temps réel embarquées de contrôle commande et de traitement du signal et des images. De la même manière, il est illusoire de penser résoudre ce type de problème avec un circuit intégré unique dont la surface est limitée. Des architectures parallèles (dans le sens où elles font intervenir plusieurs composants devant communiquer) sont nécessaires, autant pour respecter les contraintes temps réel (répartition de la charge de calcul) que pour prendre en compte la nature souvent distribuée des informations à traiter (capteurs et actionneurs délocalisés, données distribuées). Ces architectures hétérogènes, construites à partir de différents types de composants programmables (processeurs RISC, CISC, DSP) et/ou de circuits intégrés spécialisés non programmables (ASIC, FPGA, circuits full-custom), connectés à travers un réseau formé de différents types de moyens de communication (lien point-à-point série ou parallèle, bus multi-points, avec ou sans capacité mémoire) sont appelées multi-composants [1]. Plus particulièrement l'expé-

rimentation effectuée mettra en œuvre un multi-DSP composé de processeur de traitement du signal TMS320C40.

2 Présentation de la méthodologie A^3

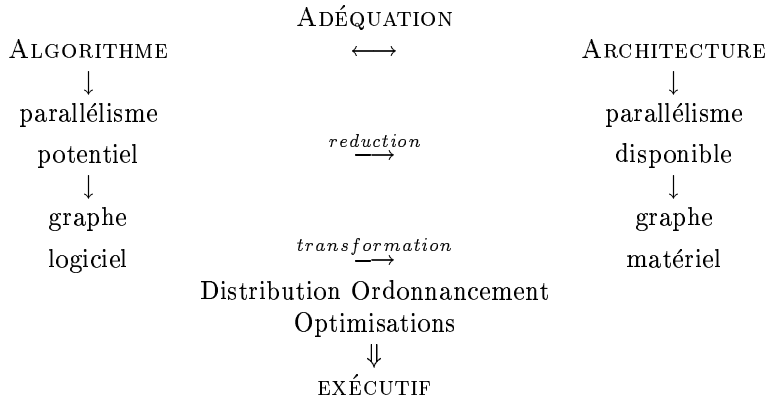
La complexité des applications temps réel embarquées nécessite à la fois des outils de spécification de haut niveau et des architectures multi-composants. Afin de réduire le nombre d'erreurs de spécification des algorithmes et de limiter au maximum les tests matériels, de nouvelles méthodes sont proposées. Elles permettent à l'utilisateur de se concentrer sur les aspects temporels qui sont cruciaux dans le domaine du temps réel (réactivité du programme et temps de réponse contraint), d'étudier les relations entre le parallélisme potentiel au niveau de l'algorithme et celui disponible au niveau de l'architecture, afin d'être déchargé de la programmation de bas niveau (exécutifs) souvent fastidieuse. La méthode A^3 - Adéquation Algorithme Architecture - est une méthode de ce type, permettant d'aider à l'implantation d'un algorithme sur une architecture donnée, conduisant éventuellement à proposer des modifications de l'architecture (dimensionnement), ou à remettre en cause l'algorithme.

L'algorithme, représenté par un graphe flot de données, possède un parallélisme potentiel. L'architecture, représentée par un graphe, possède un parallélisme effectif. L'implantation consiste, par transformations de graphes successives, à réduire le parallélisme potentiel au parallélisme effectif. Ceci est formalisé au chapitre 3. Ces transformations représentent une distribution (allocation spatiale) et un ordonnancement (allocation temporelle) des calculs sur les processeurs et des communications sur les liaisons physiques inter-processeurs.

Le terme **Adéquation** sous-entend une mise en correspondance efficace du graphe de l'algorithme, appelé **graphe logiciel** et du graphe de l'architecture, appelé **graphe matériel**.

L'adéquation consiste à choisir parmi toutes les transformations possibles une transformation permettant de respecter les contraintes temps réel tout en minimisant les composants utilisés. C'est à partir de cette distribution et de cet ordonnancement qu'un exécutif distribué temps réel, permettant l'exécution de l'algorithme sur l'architecture, peut être généré par le logiciel SynDEx qui supporte cette méthodologie.

Etant donnés un graphe logiciel et un graphe matériel, on comprend facilement qu'il y a un grand nombre de transformations possibles. Le critère utilisé par les heuristiques [2] pour déterminer, parmi toutes les transformations la transformation efficace, est lié en premier lieu aux aspects temps réel, et en particulier à la latence qu'il est primordial de maîtriser dans le cas des systèmes réactifs réalisant de la commande de processus. L'adéquation consiste à minimiser une fonction de coût, dépendant des temps d'exécution des opérations et des temps de transfert de données entre opérations. On peut imaginer que la fonction de coût fasse intervenir d'autres paramètres tels que la mémoire nécessaire pour exécuter une opération ou un transfert de données, la consommation électrique ...etc. Ces durées font partie de la caractérisation du modèle matériel. Cette caractérisation consiste à associer à chaque composant du graphe matériel l'ensemble des opérations qu'il est capable de réaliser, puis à



chaque opération on associe une liste de caractéristiques contenant sa durée d'exécution, la mémoire nécessaire, la consommation ... etc.

La distribution consiste à effectuer une partition du graphe logiciel en autant, ou moins d'éléments de partition que le graphe matériel contient de nœuds. Faire une partition du graphe logiciel revient à distribuer les nœuds du graphe logiciel sur les nœuds du graphe matériel, donc à affecter des sous-graphes du graphe logiciel à des nœuds du graphe matériel. A chaque sous-graphe est associé un ordre partiel. Le nombre d'ordre partiel obtenu est égal au cardinal de la partition choisie. Etant donné que chaque nœud du graphe matériel correspond à une machine à états finie (machine séquentielle), il faut y ordonnancer les nœuds du graphe logiciel qui lui ont été affectés. Cela revient à linéariser (rendre total) l'ordre partiel associé à chaque sous-graphe du graphe logiciel.

La recherche de la distribution et de l'ordonnancement les plus efficaces est un problème NP-complet i.e. la solution optimale ne peut être obtenue sûrement qu'en énumérant toutes les solutions. Cela prend un temps prohibitif dès que le problème posé n'est plus trivial (concrètement dès que les graphes logiciel et matériel ont plus d'une dizaine de nœuds). En d'autres termes, il n'existe pas d'algorithme s'exécutant en un temps polynomial pour trouver la solution optimale. On a généralement recours à des heuristiques qui donnent des solutions approchées, sous-optimales. SynDEx possède une heuristique qui propose donc une solution de ce type. La solution proposée par SynDEx peut correspondre dans certains cas à la solution optimale mais on ne le saura jamais puisqu'on n'aura pas évalué toutes les solutions.

3 Formalisation de l'implantation avec des graphes

Nous avons choisi des modèles de graphes pour unifier la spécification de l'architecture et de l'algorithme puis pour décrire l'implantation de l'algorithme sur l'architecture en termes de transformations sur ces graphes. Une transformation est une composition de relations sur

des couples de graphes. Le modèle d'architecture sert principalement à étiqueter les nœuds et les arcs du graphe qui décrit l'algorithme afin de pouvoir sélectionner parmi toutes les transformations de graphes, une transformation intéressante suivant un critère. Par exemple on peut rechercher les chemins critiques et les boucles critiques lorsqu'on s'intéresse à l'étiquette durée quand on cherche à optimiser les performances temps réel.

3.1 Architecture : graphe matériel

L'architecture est modélisée par un graphe non orienté représentant un réseau de processeurs MIMD (chaque processeur effectue son programme sur ses propres données) ou SPMD (plusieurs processeurs effectuent le même programme sur des données différentes), dont chaque sommet est un processeur et chaque arc est une liaison physique de communication bidirectionnelle qui permet des transferts de données entre les mémoires des processeurs, au besoin par l'intermédiaire d'une mémoire commune.

Dans ce rapport, on a choisi de modéliser des architectures n'ayant que des liaisons point à point, on ne considère pas le cas des liaisons multi-points (bus, mémoire partagée,...).

Un processeur comprend une unité de calcul, une unité d'interfaçage avec l'environnement (E/S), une unité de communication pour chaque arc adjacent, une unité de mémoire partagée.

Un graphe matériel G_m est un couple (P, L) où :

- P est l'ensemble des *nœuds* processeurs,
- $L \subseteq P \times P$ est l'ensemble des arêtes, couples non ordonnés de sommets, qui représentent des liens physiques de communications bidirectionnelles inter-processeurs.
  On a : $L = \{l_{ij}\}_{i<j} = \{(p_i, p_j) \text{ tq } i < j\} \Longleftrightarrow [p_i \mathcal{R} p_j]$ où $\mathcal{R}$, est la relation symétrique " est en communication directe avec " associée à L.

G_m est un graphe non orienté.

Remarque 1 *Ce graphe matériel est dit* encapsulé.

Exemple

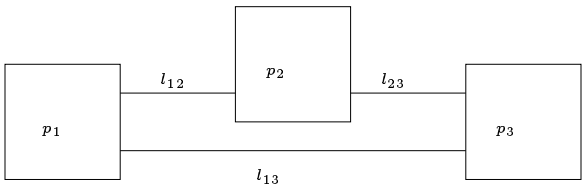


FIG. 1 – *Graphe matériel encapsulé*

$P = \{p_1, p_2, p_3\}$
$L = \{l_{12}, l_{23}, l_{13}\}$
avec :
$l_{12} = (p_1, p_2)$
$l_{23} = (p_2, p_3)$
$l_{13} = (p_1, p_3)$

Chaque *nœud* processeur p, correspond au graphe (S_p, c_p) où :

- S_p est l'ensemble des unités de calcul et des unités de communication du processeur p, on a :
  $S_p = S_{p,cal} \cup S_{p,com}$ où $S_{p,cal}$ désigne l'unité de calcul du processeur p et $S_{p,com} = \bigcup_j S_{p,com^j}$ désigne les unités de communication de ce même processeur.
- $c_p \subseteq (S_{p,cal} \times S_{p,com}) \cup (S_{p,com} \times S_{p,com}) \cup (S_{p,com} \times S_{p,cal})$, est l'ensemble des liens intra-processeurs du processeur p mais inter-unités (unité de calcul-unité de communication, unité de communication-unité de communication).
  c_p peut se décomposer en deux sous-ensembles :
  - c_p^*: ensemble des liens inter-unités de calcul et de communication.
  - c_p' : ensemble des liens inter-unités de communication.

  Ces liens sont bidirectionnels.

Remarque 2 *Ce graphe matériel* $((S, C), L)$ *est dit* développé *(cf. exemple figure 2).*
On a :
$S = S_{cal} \cup S_{com}$ *avec* $S_{cal} = \bigcup_{p \in P} S_{p,cal}$ *et* $S_{com} = \bigcup_{p \in P} S_{p,com} = \bigcup_{p \in P} (\bigcup_j S_{p,com^j})$
$C = \bigcup_{p \in P} c_p = \bigcup_{p \in P} c_p^* \cup c_p'$
$L \subseteq S_{com} \times S_{com}$ *et plus précisément :*
$\forall p_i, p_j \in P, p_i \neq p_j$, *on a :* $l_{ij} \subseteq S_{p_i,com} \times S_{p_j,com}$ *ce qui signifie que les processeurs ne peuvent communiquer entre eux que grâce aux unités de communication propres à chacun d'eux.*

Remarque 3 *Le graphe* développé *sera utilisé lors de la transformation* communication *(cf. §3.3.2), afin de définir avec précision les communications inter-processeurs.*

Auparavant, il est préférable d'utiliser le modèle encapsulé *car le modèle* développé *n'apporte rien. En effet, le* routage *concerne l'inventaire des combinaisons de liens inter-processeurs et donc mentionner des arcs intra-processeurs n'est pas utile. La* partition *concerne l'allocation* spatiale *des opérations du graphe logiciel sur les sommets du graphe matériel et l'allocation* spatiale *des dépendances de données sur les arcs inter-processeurs. Il est implicite que les opérations, qui sont des calculs, ne peuvent être effectuées que sur des unités de calcul et que les transferts de données entre opérations placées sur des processeurs différents s'effectuent grâce à des liens inter-processeurs et des unités de communication.*

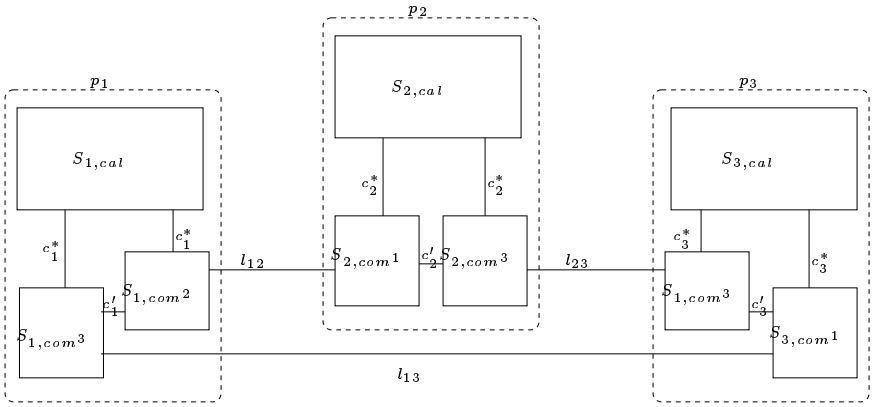


FIG. 2 – *Graphe matériel développé*

3.2 Algorithme : graphe logiciel

L'algorithme est modélisé par un graphe flot de données conditionné (graphe orienté). Chaque sommet de graphe (nœud) représente une opération de calcul, d'entrée-sortie, de mémorisation ou de conditionnement.

Chaque arc reliant deux nœuds traduit :

- un ordre partiel sur les opérations à réaliser encore appelé **ordre d'exécution** ;
- un transfert itératif de données (**flot de données**), établissant une précédence entre deux actions.

L'ensemble des arcs ainsi définis et l'ensemble des nœuds associés forment un graphe de dépendances des données ou encore un graphe flot de données. Les nœuds sans prédécesseurs (resp. sans successeurs) sont les nœuds d'ENTREE (resp. de SORTIE) ; ils représentent l'interface avec l'environnement.

Ce modèle met en évidence le parallélisme potentiel de l'algorithme (ordre partiel induit par les précédences du graphe) et la mémoire d'état. Cette dernière est représentée par l'ensemble des nœuds particuliers (notés par un \$) qui permettent d'une part de mémoriser un élément du flot de données d'une exécution à l'autre du graphe et d'autre part de "casser" les circuits dans le graphe flot de données. Le graphe flot de données est réexécuté chaque fois qu'une donnée se présente sur un nœud d'ENTREE.

Un graphe logiciel G_l est un graphe orienté sans circuit.

G_l est un couple (O, D) où :

- O est l'ensemble des opérations à exécuter, Card $O = n_o$.

– $D \subseteq O \times O$ est l'ensemble des arcs (o_i, o_j), couples ordonnés de sommets, qui traduisent d'une part l'ordre dans lequel les opérations vont être exécutées et d'autre part la dépendance de données d'une opération à l'autre, ce qui se traduira lors de l'exécution par un transfert de données. On a :
$D = \{d_{ij}\} = \{(o_i, o_j)\} \Longleftrightarrow o_i \preceq o_j$ avec : $\preceq$ =" est exécuté avant". D définit donc une relation d'ordre partiel sur l'exécution des opérations (notée $\preceq$) irréflexive, antisymétrique et transitive puisque G_l est un graphe sans circuit.
Certaines opérations, qui ne s'échangent pas de données, donc qui ne sont pas dépendantes, vont pouvoir être exécutées en parallèles si les ressources matérielles sont disponibles. Dans ce cas il n'y a pas de relation d'ordre entre l'exécution de ces opérations. Pour ces couples $(o_i, o_j) \in O \times O$, on a : $o_i \not\preceq o_j$ et $o_j \not\preceq o_i$. Donc O, est un ensemble partiellement ordonné par la relation $\preceq$.

Exemple associé à la figure 3

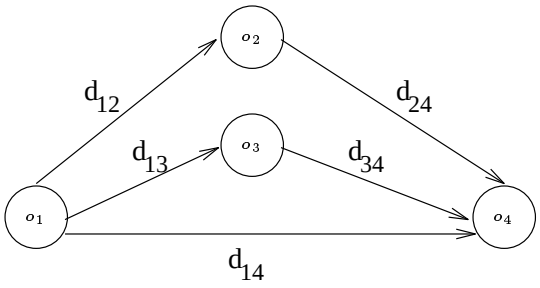


FIG. 3 – *Graphe logiciel*

$O = \{o_1, o_2, o_3, o_4\}$
$D = \{d_{12}, d_{13}, d_{14}, d_{24}, d_{34}\}$
avec :
$d_{12} = (o_1, o_2) \Longleftrightarrow o_1 \preceq o_2$
$d_{13} = (o_1, o_3) \Longleftrightarrow o_1 \preceq o_3$
$d_{14} = (o_1, o_4) \Longleftrightarrow o_1 \preceq o_4$
$d_{24} = (o_2, o_4) \Longleftrightarrow o_2 \preceq o_4$
$d_{34} = (o_3, o_4) \Longleftrightarrow o_3 \preceq o_4$
$o_2 \not\preceq o_3$ donc o_2 pourra s'exécuter en parallèle avec o_3 si on a les ressources nécessaires.

3.3 Implantation : transformations de graphes

L'implantation du graphe logiciel sur le graphe matériel est formalisé en termes des 3 transformations de graphes suivantes : *routage*, *distribution* et *ordonnancement*.

3.3.1 Routage

Le *routage* est une transformation du graphe matériel (P, L). Le but de cette transformation est de définir les différentes *routes* permettant de communiquer d'un processeur à l'autre, lorsque ceux-ci ne sont pas reliés. Cela revient à transformer le graphe matériel afin qu'il soit complètement connecté. Bien que le graphe matériel soit nécessairement connexe, certains processeurs ne sont pas forcément reliés entre eux par des liens directs ou pas uniquement. Autrement dit, pour communiquer d'un processeur à un autre, on peut utiliser le lien direct (s'il existe) ou passer par un(des) processeur(s) intermédiaire(s).

On appelle R, l'ensemble de toutes les *routes* (ou chaînes) qui sont sans cycle dans (P, L). R est donc une séquence d'arcs $\{u_1, ..., u_q\}(u_1 \neq u_q)$ tel que chaque arc u_r de la séquence $(2 \leq r \leq q-1)$ ait une extrémité commune avec l'arc $u_{r-1}(u_{r-1} \neq u_r)$ et l'autre extrémité commune avec l'arc $u_{r+1}(u_{r+1} \neq u_r)$.

Remarque 4 *La route $r \in R$ permettant de passer du processeur p_I au processeur p_{II} en utilisant n processeurs intermédiaires $p_1, ..., p_n$ est donc définie par :*

- *Dans le modèle* encapsulé *:*
  $r = (l_{I1}, l_{12}, ..., l_{nII})$
- *Dans le modèle* développé *:*
  $r = (c^*_I, l_{I1}, c'_1, l_{12}, c'_2, ..., c'_n, l_{nII}, c^*_{II})$

Remarque 5 *Si l'on considère le modèle matériel* encapsulé, *L est un sous-ensemble de couples $P \times P$, et si l'on considère le modèle matériel* développé, *L est un sous-ensemble de couples $S_{com} \times S_{com}$.*

On définit le *routage* de la manière suivante :
Soit $\mathcal{G}_o$, l'ensemble des graphes orientés et $\mathcal{G}_{no}$, l'ensemble des graphes non orientés.

$$\begin{array}{lccc} \text{routage :} & \mathcal{G}_o \times \mathcal{G}_{no} & \longrightarrow & \mathcal{G}_o \times \mathcal{G}_{no} \\ & ((O,D),(P,L)) & \overset{routage}{\longrightarrow} & ((O,D),(P,R)) \end{array}$$

Remarque 6 *Le* routage *consiste à rajouter des arcs sur le graphe matériel. Ces nouveaux arcs sont des combinaisons de liens physiques existant i.e. on ne rajoute pas de liens physiques. La ressource disponible étant inchangée, un arbitrage peut être nécessaire pour gérer le partage de celle-ci.*

Exemple
$R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ avec:

$$\begin{aligned} r_1 &= l_{12} \\ r_2 &= l_{23} \\ r_3 &= l_{13} \\ r_4 &= (l_{12}, l_{23}) \\ r_5 &= (l_{12}, l_{13}) \\ r_6 &= (l_{23}, l_{13}) \end{aligned}$$

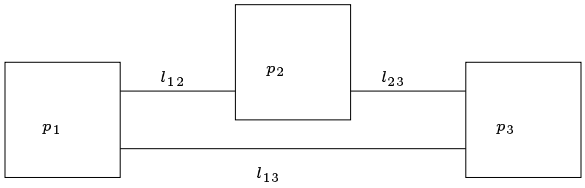


FIG. 4 – *Graphe matériel*

Autrement dit, pour communiquer entre p_1 et p_3, par exemple, deux types de communications peuvent être utilisées. D'une part le lien physique direct l_{13}, d'autre part, la route qui comprend le lien l_{12} puis le lien l_{23}.

Lorsque p_1 a deux communications à envoyer à p_3, plutôt que de les envoyer séquentiellement, i.e. une après l'autre, elles peuvent être envoyées en parallèle en utilisant $r_3 = l_{13}$ et $r_4 = (l_{12}, l_{23})$.

Remarque 7 $L \subseteq R$

Remarque 8 *Le routage correspond à la partie non réflexive de la fermeture transitive de la relation $\mathcal{R}$ " être en communication directe avec" précédemment définie.* La fermeture transitive $t(\mathcal{R})$ d'une relation $\mathcal{R}$ sur un ensemble de n éléments étant définie par :
$t(\mathcal{R}) = \bigcup_{k=1}^{n} \mathcal{R}^k$.

Remarque 9 *Le graphe matériel ayant subi la transformation de* routage *est composé de deux graphes partiels. Le premier graphe partiel est engendré par l'ensemble des arcs correspondant aux liens physiques (autrement dit le graphe de départ) et le deuxième est engendré par l'ensemble de toutes les combinaisons possibles de liens physiques permettant de définir des chaînes de longueur > 1 (i.e. non réduit à un arc) et sans cycle.*

Propriété 1 *Le routage est une opération interne de l'ensemble des couples de graphes orienté et non orienté dans l'ensemble des couples de graphes orienté et non orienté.* En effet, le graphe logiciel n'est pas modifié. De plus, le graphe matériel caractérise une machine dont les ressources sont finies i.e. le nombre de processeurs est limité ainsi que le nombre de liens inter-processeurs. Autrement dit, en termes de graphes, cela signifie que le nombre de sommets et d'arcs est fini. Donc, l'ensemble des combinaisons de tous les arcs telles que ces combinaisons définissent des chaînes sans cycle est fini et unique(cf. les algèbres de chemins dans [3]). Le *routage* est donc une fonction f_{route} de $\mathcal{G}_o \times \mathcal{G}_{no} \longrightarrow \mathcal{G}_o \times \mathcal{G}_{no}$ donc c'est une opération interne.

Notations

Soit $\mathcal{R}$, une relation $\subseteq A \times B$.

On définit l'ensemble $\mathcal{R}(a)$, pour un élément $a \in A$ fixé par :

$$\mathcal{R}(a) = \{b \in B \text{ tq } a\mathcal{R}b\}$$

routage f_{route}: $\mathcal{G}_o \times \mathcal{G}_{no} \longrightarrow \mathcal{G}_o \times \mathcal{G}_{no}$
$(G_l, G_m) \longmapsto (G_l, G'_m)$
et f_{route} peut être vue comme une relation $\mathcal{R}_{route}$

$$\boxed{\begin{array}{l}\mathcal{R}_{route} \subseteq (\mathcal{G}_o \times \mathcal{G}_{no} \times \mathcal{G}_o \times \mathcal{G}_{no}) \\ \mathcal{R}_{route}(G_l, G_m) = \{(G_l, G'_m)\}\end{array}}$$

Lors du routage, seul le graphe matériel est modifié.

3.3.2 Distribution

On appelle *distribution*, l'allocation *spatiale* des sommets du graphe logiciel aux sommets du graphe matériel et l'allocation *spatiale* des dépendances de données aux liens physiques.

A l'issue de la *distribution*, on doit pouvoir associer un nœud(resp. un arc) du graphe logiciel à un nœud(resp. un arc) du graphe matériel .

La *distribution* peut se définir sur les deux types de modèles matériels précédemment définis :

- sur le modèle matériel *encapsulé*, la distribution s'appelle *partition*,
- sur le modèle matériel *développé*, la distribution s'appelle *communication*.

On verra dans la suite, quelle relation permet de passer de la *partition* à la *communication*. On verra également que l'on peut définir la *communication* soit à partir du graphe partitionné, soit à partir du graphe logiciel de départ.

La partition

La *partition* consiste à décomposer le graphe orienté en sous-graphes disjoints sous la contrainte que le nombre de sous-graphes soit inférieur ou égal au nombre de processeurs. On réalise donc une partition des sommets du graphe logiciel et chaque élément de cette partition est affecté à un processeur qui correspond au sous-graphe engendré par les sommets appartenant à cet élément. Plus précisément, chaque sous-graphe engendré forme ce que Zwiers et Janssen appellent dans [4] une *région atomique*. Cela revient à encapsuler le graphe logiciel initial. Les arcs intra-partitions, donc intra-région atomique, correspondent à des communications intra-processeurs et chaque arc inter-partition(inter-région atomique) nécessite une communication inter-processeur qui est affectée à une route du graphe matériel construite lors du *routage*.

Soit O_p, l'ensemble des opérations du graphe logiciel affectées au processeur p appartenant à P.

$$O_p = \bigcup_{i=1}^{n_p} o_i \text{ avec } o_i \in O,\ \forall p \in P \text{ et } \sum_{p=1}^{\text{Card }(P)} n_p = \text{Card}(O) = n_o$$

$$\begin{cases} O_{p1} \cap O_{p2} = \emptyset & \forall\, p_1, p_2 \in P \\ \bigcup\limits_{p \in P} O_p = O \end{cases}$$

$D_p \subseteq O_p \times O_p \, \forall p \in P$, désigne l'ensemble de dépendances des données entre les différentes opérations appartenant à O_p. C'est donc un ensemble de liens intra-processeurs.

$D_p = \bigcup\limits_{i,j} d_{ij} \, \forall p \in P$, avec i et j tels que o_i affecté à p, o_j affecté à p.

$\forall\, d_{ij} \in D_p$, $\forall p \in P$, on a $d_{ij} = \{(o_i, o_j)\} \Longleftrightarrow o_i \preceq_p o_j$ avec $\preceq_p$= "est exécuté avant (sur un même processeur)".

$$\begin{cases} D_{p_1} \cap D_{p_2} = \emptyset & \forall\, p_1, p_2 \in P \\ \bigcup\limits_{p \in P} D_p = D_P \end{cases}$$

D_P, définit donc un ordre partiel (noté $\preceq_P$) sur l'exécution des opérations *intra*-processeurs.

$D_r \subseteq O_{p_i} \times O_{p_j} \quad \forall p_i, p_j \in P, p_i \neq p_j$, est l'ensemble des dépendances de données correspondant aux communications inter processeurs utilisant la route r appartenant à R. En effet, on peut affecter plusieurs d_{ij} à une même route r.

$D_r = \bigcup\limits_{i,j} d_{ij} \, \forall r \in R$, avec i et j tels que o_i affecté à p, o_j affecté à $p' \neq p$

$\forall d_{ij} \in D_r$, $\forall r \in R$, on a $d_{ij} = \{(o_i, o_j)\} \Longleftrightarrow o_i \preceq_r o_j$ avec $\preceq_r$= "est exécuté avant (sur 2 processeurs différents)".

$$\begin{cases} D_{r_1} \cap D_{r_2} = \emptyset & \forall\, r_1, r_2 \in R \\ \bigcup\limits_{r \in R} D_r = D_R \end{cases}$$

D_R, définit donc un ordre partiel (noté $\preceq_R$) sur l'exécution des opérations *inter*-processeurs.

$$\begin{cases} (\bigcup\limits_{p \in P} D_p) \bigcup (\bigcup\limits_{r \in R} D_r) = D \\ D_p \cap D_r = \emptyset \quad \forall p, r \in P \times R \end{cases}$$

On a donc, également par construction la propriété suivante:

Propriété 2 Les liaisons D_P et D_R forment une partition de l'ensemble D.

Proposition 1 L'ordre partiel $\preceq$ induit par les précédences du graphe logiciel de départ est conservé lors de la partition. *En effet, il est remplacé par l'union des deux ordres partiels $\preceq_P$ et $\preceq_R$ précédemment définis.*

Ainsi, le graphe logiciel (O, D) peut se mettre sous la forme:

$$(O, D) = \left(\bigcup_{p \in P} O_p, (\bigcup_{p \in P} D_p) \bigcup (\bigcup_{r \in R} D_r) \right)$$

On définit la *partition* en réécrivant (O, D) de la manière suivante :
Soit $\mathcal{G}_o$, l'ensemble des graphes orientés et $\mathcal{G}_{no}$, l'ensemble des graphes non orientés.
Soit G_{pR}, l'ensemble des graphes *partitionnés* associé au graphe logiciel (O, D).

$$\begin{array}{rccc} \text{partition :} & \mathcal{G}_o \times \mathcal{G}_{no} & \longrightarrow & \mathcal{G}_o \times \mathcal{G}_{no} \\ & ((O,D),(P,R)) & \overset{partition}{\longrightarrow} & ((G_{pR},(P,R)) \end{array}$$

$$\text{avec : } G_{pR} = \left(\bigcup_{p \in P} (O_p, D_p), \bigcup_{r \in R} D_r \right)$$

(O_p,D_p) représente un sous-graphe, engendré par O_p, ensemble des sommets de O distribué sur le processeur p. D_p, étant des liens intra-processeurs, sont bien des arcs ayant leurs deux extrémités dans O_p. Les sous-graphes obtenus ($\bigcup_{p \in P} (O_p, D_p)$), sont les sommets encapsulants autrement dit les régions atomiques précédemment définies du graphe distribué et les arcs ($\bigcup_{r \in R} D_r$) du graphe distribué sont les arcs initiaux privés des arcs figurant dans les sous-graphes.

Exemple associé à la figure 5

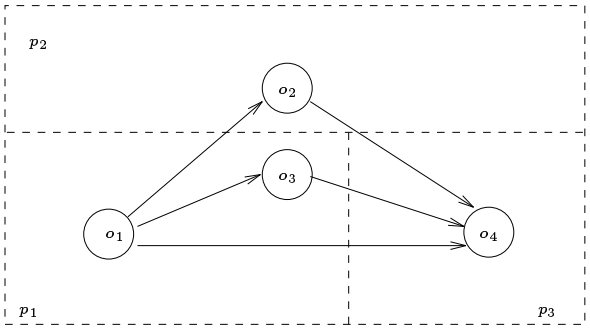


FIG. 5 – *Partition*

Dépendances intra-partition :
$O_{p_1} = \{o_1, o_3\} \Rightarrow D_{p_1} = \{d_{13}\}$
$O_{p_2} = \{o_2\} \Rightarrow D_{p_2} = \emptyset$ puisqu'il n'y a qu'une opération sur le processeur 2
$O_{p_3} = \{o_4\} \Rightarrow D_{p_3} = \emptyset$ pour la même raison
Dépendances inter-partition :

- Une seule dépendance inter-partition :
  - d_{12} entre p_1 et p_2, on l'affecte à r_1. $\longleftrightarrow D_{r_1} = D_{l_{12}} = \{d_{12}\}$
  - d_{24} entre p_2 et p_3, on l'affecte à r_2. $\longleftrightarrow D_{r_2} = D_{l_{23}} = \{d_{24}\}$

- Une double dépendance inter-partition d_{14} et d_{34} entre p_1 et p_3. Il y a plusieurs possibilités d'affecter $\{d_{14}\}$ et $\{d_{34}\}$ aux liens physiques :
  - (1) $\{d_{14}\}$ et $\{d_{34}\}$ sont effectués en parallèle sur les routes r_3 et r_4.

$$\Longrightarrow \left\{ \begin{array}{l} D_{r_3} = D_{l_{13}} = \{d_{14}\} \\ D_{r_4} = D_{(l_{12},l_{23})} = \{d_{34}\} \end{array} \right.$$

  - (2) $\{d_{14}\}$ et $\{d_{34}\}$ sont séquentialisés sur la route r_3

$$\Longrightarrow \left\{ \begin{array}{l} D_{r_3} = D_{l_{13}} = \{d_{14}, d_{34}\} \\ D_{r_4} = \emptyset \end{array} \right.$$

Remarque 10 *La partition ne modifie pas le nombre de nœuds et le nombre d'arcs du graphe logiciel. C'est donc une simple réécriture du graphe logiciel consistant à structurer le graphe en régions atomiques. Le nombre de décompositions du graphe logiciel en régions atomiques est fini puisque le graphe logiciel est composé d'un nombre fini de nœuds et d'arcs.*

Remarque 11 *Soit $G_l = (O, D)$, un graphe logiciel donné.*

Alors, on peut construire plusieurs, mais un nombre fini, graphes partionnés $G_{pR}(.)$, associés à G_l.

$$(O, D) \stackrel{partition}{\longrightarrow} (G_{pR}(1), \cdots, G_{pR}(n))$$

Ce qui peut encore s'écrire : $(G_l, G'_m)\ \mathcal{R}_{par}\ (G_{pR}(i), G'_m)$ avec $1 \leq i \leq n$ où n est le nombre de graphes partionnés et $\mathcal{R}_{par}$ est la relation " a pour graphe partionné ".

Propriété 3 *La partition est donc une relation $\mathcal{R}_{par}$ de l'ensemble des couples de graphe orienté et non orienté vers l'ensemble des couples de graphes orienté et non orienté.*

$$\begin{array}{|l} \mathcal{R}_{par} \subseteq (\mathcal{G}_o \times \mathcal{G}_{no} \times \mathcal{G}_o \times \mathcal{G}_{no}) \\ \mathcal{R}_{par}(G_l, G'_m) = \{(G_{pR}(1), G'_m), \cdots, (G_{pR}(n), G'_m)\} \end{array}$$

En effet, le graphe matériel n'est pas modifié et le nombre de décompositions du graphe logiciel en régions atomiques (cf : remarque 10) est fini.

La communication

La *communication* consiste dans un premier temps à décomposer le graphe orienté en régions atomiques sous la même contrainte que dans le cas de la *partition*. Donc, pour réaliser cette transformation, on peut soit effectuer une décomposition en régions atomiques du graphe logiciel de départ soit utiliser la décomposition de ce même graphe effectuée lors de la *partition*. Dans ce cas, le sous-graphe (O_p, D_p) engendré par O_p et affecté lors de la *partition* au processeur p du graphe matériel *encapsulé*, sera affecté lors de la *communication*, à l'unité de calcul $S_{p,cal}$ du processeur p.

Dans un deuxième temps, la *communication* permet de décrire avec plus de précision les communications *inter*-processeurs. Les dépendances de données D_R *inter*-régions atomiques du graphe logiciel sont affectées aux routes du graphe matériel définies lors du routage dans le cas du graphe matériel *développé* (cf:remarque 4). On a vu que ces routes sont des combinaisons de liens inter-processeurs et de liens intra-processeurs mais inter-unités (unité de calcul-unité de communication, unité de communication-unité de communication). Pour que cette affectation soit possible, des nœuds représentant des opérations de communication sont rajoutés au graphe logiciel et sont affectés aux unités de communication des processeurs composant le graphe matériel. Ainsi, un nœud(resp. un arc) du graphe logiciel sera affecté à un nœud(resp. un arc) du graphe matériel (cf: tableau page 21).

Remarque 12 *Chaque arc inter-processeur est remplacé par un graphe. Ce graphe est muni d'une relation d'ordre total $<$ donc l'ordre partiel du graphe logiciel de départ ne sera pas modifié.*

En supposant que la route $r \in R$ permet de passer du processeur p_I au processeur p_{II} en utilisant n processeurs intermédiaires $p_1, ..., p_n$ (cf: remarque 4), chaque $d_{ij} \in D_r$ est défini par le graphe linéaire suivant :

$\forall r \in R, \quad \forall d_{ij} \in D_r,$

$$d_{ij} \overset{communication}{\longrightarrow} \left(d_{c_I^*}, O_{r,1}, d_{l_{I1}}, O_{r,2}, d_{c_1'}, O_{r,3}, d_{l_{12}} ..., O_{r,2n+1}, d_{l_{nII}}, O_{r,2n+2}, d_{c_{II}^*}\right)$$

L'arc $d_{c_I^*}$(resp. $d_{c_{II}^*}$) sera affecté à l'arc *intra*-processeur mais *inter*-unités c_I^*(resp. c_{II}^*) du processeur p_I(resp. p_{II}).

Chaque nœud $O_{r,i}$ est une opération de communication. Les nœuds $O_{r,2i}$ et $O_{r,2i-1}$ avec $i > 0$ seront affectés aux unités de communication S_{p,com^i} du ième processeur composant la route r. Le nombre de nœuds $O_{r,i}$ ainsi rajoutés est égal au double du nombre de processeurs intermédiaires plus deux qui représentent le processeur émetteur p_I et le processeur récepteur p_{II}. On pose: $\bigcup_i O_{r,i} = O_r$

Enfin, les arcs de type $d_{l_{ij}}$ sont affectés aux liens *inter*-processeurs l_{ij} du graphe matériel.

Ainsi, le graphe logiciel (O, D) peut se mettre sous la forme :

$$(O, D) = \left(\bigcup_{p \in P} (O_p, D_p) \cup \bigcup_{r \in R} O_r, \bigcup_{p \in P} (d_{c_p^*} \cup d_{c_p'}) \cup \bigcup_{l \in L} d_l \right)$$

On définit la *communication* en réécrivant (O, D) de la manière suivante :

Soit $\mathcal{G}_o$, l'ensemble des graphes orientés et $\mathcal{G}_{no}$, l'ensemble des graphes non orientés. Soit G_{cR}, l'ensemble des graphes distribués sur le graphe matériel développé, associé au graphe logiciel (O, D).

$$\begin{array}{lccc} \text{communication :} & \mathcal{G}_o \times \mathcal{G}_{no} & \longrightarrow & \mathcal{G}_o \times \mathcal{G}_{no} \\ & ((O, D), (P, R)) & \overset{communication}{\longrightarrow} & ((G_{cR}, (P, R)) \end{array}$$

$$\text{avec: } G_{cR} = \left((\bigcup_{p\in P} O_p \cup \bigcup_{r\in R} O_r, \bigcup_{p\in P} (D_p \cup d_{c'_p} \cup d_{c^*_p})) \ , \ \bigcup_{l\in L} d_l \right)$$

$(\bigcup_{p\in P} O_p \cup \bigcup_{r\in R} O_r, \bigcup_{p\in P} (D_p \cup d_{c'_p} \cup d_{c^*_p}))$ représentent les sommets du graphe logiciel distribué. Ils correspondent aux sous-graphes engendrés par $(\bigcup_{p\in P} O_p \cup \bigcup_{r\in R} O_r)$. Les arcs D_p, $d_{c^*_p}$ et $d_{c'_p}$ étant des arcs correspondant à des liens intra-processeurs sont bien des arcs ayant leurs deux extrémités dans $(\bigcup_{p\in P} O_p \cup \bigcup_{r\in R} O_r)$. Les arcs $\bigcup_{l\in L} d_l$ sont les arcs du graphe logiciel distribué et correspondent aux liens inter-processeurs.

Exemple associé à la figure 6:

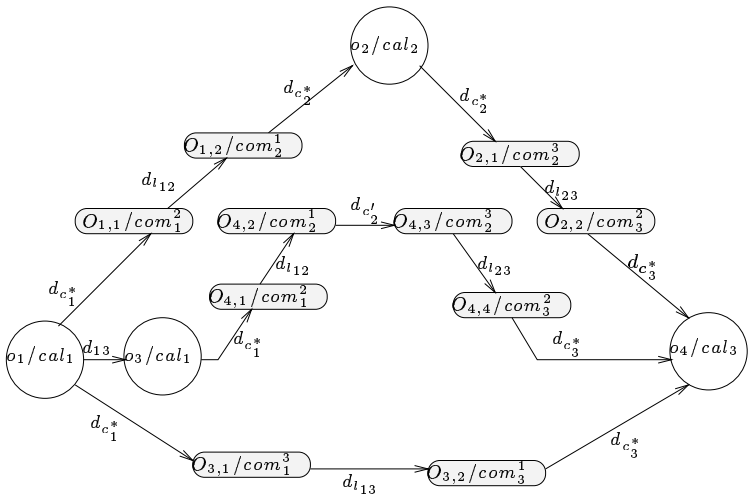


FIG. 6 – *Communication*

o_i/cal_j avec $1 \leq i \leq 4$ et $1 \leq j \leq 3$ indique que l'opération o_i du graphe logiciel va s'effectuer sur l'unité de calcul du processeur j.

$O_{i,j}/com^l_k$ avec $1 \leq i,j \leq 4$ et $1 \leq k,l \leq 3$ est la $j$ème opération de la route i affectée à l'unité de communication du processeur k permettant de communiquer avec le processeur l.

Ainsi on peut mettre en évidence la relation entre les arcs inter-processeurs du graphe distribué et entre les arcs de communication intra-processeurs mais inter-unités et les arcs de communications inter-processeurs correspondant à un transfert de données du graphe après communication.

$D_{r_1} = \{d_{12}\} = \{d_{c^*_1}, O_{1,1}, d_{l_{12}}, O_{1,2}, d_{c^*_2}\}$
$D_{r_2} = \{d_{24}\} = \{d_{c^*_2}, O_{2,1}, d_{l_{23}}, O_{2,2}, d_{c^*_3}\}$

$D_{r_3} = \{d_{14}\} = \{d_{c_1^*}, O_{3,1}, d_{l_{13}}, O_{3,2}, d_{c_3^*}\}$
$D_{r_4} = \{d_{34}\} = \{d_{c_1^*}, O_{4,1}, d_{l_{12}}, O_{4,2}, d_{c_2'}, O_{4,3}, d_{l_{23}}, O_{4,4}, d_{c_3^*}\}$

Proposition 2 *La communication ne modifie pas l'ordre partiel du graphe logiciel.* En effet, cette transformation a uniquement consisté à rajouter des arcs intra-processeurs et des opérations de communication pour que l'on puisse distinguer les unités de calcul des unités de communication.

Remarque 13 *Soit $G_l = (O, D)$, un graphe logiciel donné.*

Alors, on peut construire plusieurs, mais un nombre fini, graphes distribués $G_{cR}(.)$ sur le modèle matériel développé, associés à G_l.

$$(O, D) \overset{communication}{\longrightarrow} (G_{cR}(1), \cdots, G_{cR}(n))$$

Ce qui peut encore s'ecrire : $(G_l, G'_m)\, \mathcal{R}_{com}\, (G_{cR}(i), G'_m)$ avec $1 \leq i \leq n$ où n est le nombre de graphes distribués sur le modèle développé et $\mathcal{R}_{com}$ est la relation " a pour graphe distribué sur le modèle développé ".

Propriété 4 *La communication est donc une relation $\mathcal{R}_{com}$ de l'ensemble des couples de graphe orienté et non orienté vers l'ensemble des couples de graphe orienté et non orienté.*

$$\boxed{\begin{array}{l} \mathcal{R}_{com} \subseteq (\mathcal{G}_o \times \mathcal{G}_{no} \times \mathcal{G}_o \times \mathcal{G}_{no}) \\ \mathcal{R}_{com}(G_l, G'_m) = \{(G_{cR}(1), G'_m), \cdots, (G_{cR}(n), G'_m)\} \end{array}}$$

En effet, le graphe matériel n'est pas modifié et le nombre de décompositions du graphe logiciel en régions atomiques (cf : remarque 10) est fini.

Distribution

On définit la distribution en réécrivant (O, D) de la manière suivante :
Soit $\mathcal{G}_o$, l'ensemble des graphes orientés et $\mathcal{G}_{no}$, l'ensemble des graphes non orientés.
Soit $\mathcal{G}_{dR}$, l'ensemble des graphes distribués, associé à (O, D).

$$\begin{array}{lccc} \text{distribution :} & \mathcal{G}_o \times \mathcal{G}_{no} & \longrightarrow & \mathcal{G}_o \times \mathcal{G}_{no} \\ & ((O, D), (P, R)) & \overset{distribution}{\longrightarrow} & ((G_{dR}, (P, R)) \end{array}$$

$$\text{avec : } G_{dR} = \begin{cases} G_{pR} & \text{si l'on considère l'allocation spatiale de } (O, D) \\ & \text{sur le graphe matériel encapsulé} \\ G_{cR} & \text{si l'on considère l'allocation spatiale de } (O, D) \\ & \text{sur le graphe matériel développé} \end{cases}$$

Remarque 14 *Il y a le même nombre de possibilités de distribuer le graphe logiciel sur le modèle matériel encapsulé et sur le modèle matériel développé.*

- *une seule unité de calcul par processeur donc c'est raisonnable d'identifier l'un à l'autre,*

– *les nœuds opérations de communication sont rajoutés au graphe logiciel une fois la décomposition en régions atomiques effectuées et ne modifient pas l'ordre partiel du graphe initial.*

Tableau des correspondances de distribution entre graphe matériel et graphe logiciel

$G_m = ((S, C), L)$	$G_l = (O, D)$
S_{cal}	$\bigcup_{p \in P} (O_p, D_p)$
S_{com}	$\bigcup_{r \in R} O_r$
C^*	$\bigcup_{p \in P} d_{c_p^*}$
C'	$\bigcup_{p \in P} d_{c'_p}$
L	$\bigcup_{l \in L} d_l$

Propriété 5 *La distribution est donc une relation $\mathcal{R}_{dis}$ de l'ensemble des couples de graphe orienté et non orienté vers l'ensemble des couples de graphe orienté et non orienté.*

$$\mathcal{R}_{dis} \subseteq (\mathcal{G}_o \times \mathcal{G}_{no} \times \mathcal{G}_o \times \mathcal{G}_{no})$$
$$\mathcal{R}_{dis}(G_l, G'_m) = \{(G_{dR}(1), G'_m), \cdots, (G_{dR}(n), G'_m)\}$$

avec :

$$\mathcal{R}_{dis} = \begin{cases} \mathcal{R}_{par} & \textit{si l'on considère l'allocation spatiale de } (O, D) \\ & \textit{sur le graphe matériel encapsulé} \\ \mathcal{R}_{com} & \textit{si l'on considère l'allocation spatiale de } (O, D) \\ & \textit{sur le graphe matériel développé} \end{cases}$$

3.3.3 Ordonnancement

On appelle *ordonnancement*, l'allocation *temporelle* des sommets du graphe logiciel *distribué* aux sommets du graphe matériel *développé* et l'allocation *temporelle* des dépendances de données aux liens physiques.

Cette transformation consiste à étudier les relations d'ordre de tous les sous-graphes engendrés par les opérations de calcul ou de communication contraintes à être exécutées sur des unités de calcul ou de communication. Chaque unité est un automate purement séquentiel. La relation associée à chaque automate doit donc être totale à l'issue de l'*ordonnancement* et elle doit comprendre l'ordre partiel associé aux arcs de dépendances de données du graphe logiciel de départ.

L'ordonnancement va donc consister à rajouter des arcs au graphe logiciel distribué. Ces nouveaux arcs, contrairement aux arcs initiaux vont représenter simplement les précédences d'exécution du graphe logiciel distribué et non plus des transferts de données.

Renforcement de l'ordre partiel

Sur l'unité de calcul (resp : l'unité de communication) de chaque processeur composant le graphe matériel *développé*, plusieurs cas de figure peuvent se produire :

- il existe un lien direct entre 2 opérations de calcul O_p ou de communication O_r : a et b. Dans ce cas, l'ordre est "déjà" *total*. Il n'y a pas d'arcs à rajouter.
  $\Longrightarrow$ pas d'ordonnancement,
- il n'existe pas de liens directs entre une opération a et une opération b, alors :
  - si l'on peut construire un chemin sans circuit obtenu par la fermeture transitive de la relation $\preceq$, à la puissance correspondant au nombre d'arcs séparant les 2 opérations, alors on dit que a et b sont *logiquement dépendants*[5] i.e. l'une des tâches doit être achevée pour que l'autre puisse débuter.
    $\Longrightarrow$ l'ordonnancement est imposé par les précédences du graphe logiciel de départ,
  - si l'on ne peut pas construire de chemin entre a et b, a et b sont dits *logiquement indépendants*[5] ou encore *concurrents* [4]. Cependant, comme a et b doivent être exécutés de manière séquentielle, on doit les rendre *causalement dépendants*[5] i.e. on force une tâche à précéder l'autre. Ce choix est fait en prenant en compte les caractéristiques temporelles du graphe utilisées lors de l'optimisation.
    $\Longrightarrow$ l'ordonnancement est libre.

Ainsi, les arcs rajoutés vont être des arcs intra-processeurs et intra-unités (de communication ou de calcul).

- **Sur les unités de calcul**
  On a vu que D_p est l'ensemble des arcs intra-processeur et intra-unité associé à l'unité de calcul du processeur p. L'ordre partiel associé est $\preceq_p$. A l'issue de l'ordonnancement, on aura des arcs du même type sur les unités de calcul. On appelle $\bar{D}_p$, l'ensemble des arcs associés à l'unité de calcul du processeur p. $\bar{D}_p$ comprend D_p plus les nouveaux arcs qui permettent de passer de l'ordre partiel $\preceq_p$ à l'ordre total $\prec_p$.
  $D_p \subseteq \bar{D}_p \Longleftrightarrow \preceq_p \subseteq \prec_p$

  Remarque 15 *Il se peut que la relation d'ordre sur l'unité de calcul du processeur p soit déjà totale. Dans ce cas, $\bar{D}_p = D_p$.*

- **Sur les unités de communication**
  On appelle $\bar{d}_{c_p^{p'}}$, l'ensemble des arcs associés à l'unité de communication du processeur p permettant de communiquer avec le processeur p'. La relation d'ordre associée aux arcs $\bar{d}_{c_p^{p'}}$ est totale.

Ainsi, l'ensemble des graphes *distribués* G_{dR} associé au graphe logiciel (O, D) peut se mettre sous la forme :

$$G_{dR} = \left(((\bigcup_{p \in P} O_p) \cup (\bigcup_{r \in R} O_r)), \bigcup_{p \in P} (\bar{D}_p \cup d_{c'_p} \cup d_{c^*_p} \cup \bar{d}_{c_p^{p'}}) \cup (\bigcup_{l \in L} d_l) \right)$$

On définit l'ordonnancement en réécrivant G_{dR} de la manière suivante :

Soit $\mathcal{G}_o$, l'ensemble des graphes orientés et $\mathcal{G}_{no}$, l'ensemble des graphes non orientés.

Soit G_S, l'ensemble des graphes ordonnancés associé à l'ensemble des graphes distribués G_{dR} du graphe logiciel (O, D).

$$\begin{array}{rccc} \text{ordonnancement :} & \mathcal{G}_o \times \mathcal{G}_{no} & \longrightarrow & \mathcal{G}_o \times \mathcal{G}_{no} \\ & (G_{dR}, (P, R)) & \overset{ordonnancement}{\longrightarrow} & ((G_S, (P, R)) \end{array}$$

$$\text{avec : } G_S = \left((\bigcup_{p \in P} O_p \cup \bigcup_{r \in R} O_r, \bigcup_{p \in P} (\bar{D}_p \cup d_{c'_p} \cup d_{c^*_p} \cup \bar{d}_{c^{p'}_p})) \ , \ \bigcup_{l \in L} d_l \right)$$

Les sommets du graphe logiciel ordonnancé sont représentés par $(\bigcup_{p \in P} O_p \cup \bigcup_{r \in R} O_r, \bigcup_{p \in P} (\bar{D}_p \cup d_{c'_p} \cup d_{c^*_p} \cup \bar{d}_{c^{p'}_p})$. Ils correspondent aux sous-graphes engendrés par $(\bigcup_{p \in P} O_p \cup \bigcup_{r \in R} O_r)$. Les arcs $\bar{D}_p$, $d_{c^*_p}$, $d_{c'_p}$ et $d_{c^{p'}_p}$ correspondant à des liens intra-processeurs sont bien des arcs ayant leurs deux extrémités dans $(\bigcup_{p \in P} O_p \cup \bigcup_{r \in R} O_r)$. Les arcs $\bigcup_{l \in L} d_l$ sont les arcs du graphe logiciel distribué et correspondent aux liens inter-processeurs.

Remarque 16 *La relation d'ordre associée au graphe logiciel ordonnancé comprend :*

- *autant de relations d'ordre total qu'il y a d'unités de calcul et de communication. Ces relations d'ordre sont associées aux arcs $\bar{D}_p$ et $\bar{d}_{c^{p'}_p}$,*
- *des relations d'ordre au moins partiel entre ces unités. Ces relations d'ordre sont associées aux arcs $d_{c'_p}$, $d_{c^*_p}$ et d_l et ce sont ces arcs qui définissent le parallélisme potentiel de l'algorithme.*

Exemple associé à la figure 7:

Sur les unités de calcul :

1. Du processeur 1:
   $O_{p_1} = \{o_1, o_3\}$ et $D_{p_1} = (o_1, o_3) \Longleftrightarrow o_1 \prec o_3 \Longrightarrow$ Pas d'ordonnancement
2. Du processeur 2:
   $O_{p_2} = \{o_2\}$ Une seule action $\Longrightarrow$ Pas d'ordonnancement
3. Du processeur 3:
   $O_{p_3} = \{o_3\}$ Même cas que (2).

Sur les unités de communication :

4. Du processeur 1 permettant de communiquer avec le processeur 2:
   $O_{S_1^2} = \{O_{1,1}, O_{4,1}\}$

   $O_{1,1}$ et $O_{4,1}$ sont logiquement indépendants puisqu'on ne peut pas construire de chemin sans circuit par fermeture transitive des arcs existants. on doit les rendre causalement dépendants.

   $\Longrightarrow$ L'ordonnancement est libre.

   On choisit de rajouter l'arc $\bar{d}_{c_1^2} = (O_{1,1}, O_{4,1})$

5. Du processeur 1 permettant de communiquer avec le processeur 3:
   $O_{S_1^3} = \{O_{3,1}\} \Longrightarrow$ Pas d'ordonnancement

6. Du processeur 2 permettant de communiquer avec le processeur 1:
   $O_{S_2^1} = \{O_{1,2}, O_{4,2}\}$ Même cas que (4).

   On choisit de rajouter l'arc $\bar{d}_{c_2^1} = (O_{1,2}, O_{4,2})$

7. Du processeur 2 permettant de communiquer avec le processeur 3:
   $O_{S_2^3} = \{O_{2,1}, O_{4,3}\}$ Même cas que (4).

   On choisit de rajouter l'arc $\bar{d}_{c_2^3} = (O_{2,1}, O_{4,3})$

8. Du processeur 3 permettant de communiquer avec le processeur 1:
   $O_{S_3^1} = \{O_{3,2}\} \Longrightarrow$ Pas d'ordonnancement

9. Du processeur 3 permettant de communiquer avec le processeur 2:
   $O_{S_3^2} = \{O_{2,2}, O_{4,4}\}$ Même cas que (4).

   On choisit de rajouter l'arc $\bar{d}_{c_3^2} = (O_{2,2}, O_{4,4})$

Remarque 17 *Soit $G_{dR}(i)$, le ième graphe distribué associé au graphe logiciel (O, D). Alors on peut construire plusieurs, mais un nombre fini, graphes ordonnancés $G_S(i,.)$, associés à $G_{dR}(i)$.*

$$(G_{dR}(i)) \overset{ordonnancement}{\longrightarrow} (G_S(i,1), \cdots, G_S(i,N_i))$$

Ce qui peut encore s'ecrire : $(G_{dR}(i), G'_m)\ \mathcal{R}_{ord}\ (G_S(i,j), G'_m)$ avec $1 \leq j \leq N_i$ où N_i est le nombre de graphes ordonnancés associés au graphe distribué $G_{dR}(i)$ et $\mathcal{R}_{ord}$ est la relation " a pour graphe ordonnancé sur le modèle développé ".

Propriété 6 *L'ordonnancement est donc une relation $\mathcal{R}_{ord}$ de l'ensemble des couples de graphe orienté et non orienté vers l'ensemble des couples de graphe orienté et non orienté.*

$$\boxed{\begin{array}{l} \mathcal{R}_{ord} \subseteq (\mathcal{G}_o \times \mathcal{G}_{no} \times \mathcal{G}_o \times \mathcal{G}_{no}) \\ \mathcal{R}_{ord}(G_{dR}(i), G'_m) = \{(G_S(i,1), G'_m), \cdots, (G_S(i,N_i), G'_m)\} \end{array}}$$

En effet, le graphe matériel n'est pas modifié et le nombre d'ordonnancements associés à un graphe logiciel distribué est fini puisque le nombre de nœuds et d'arcs est fini.

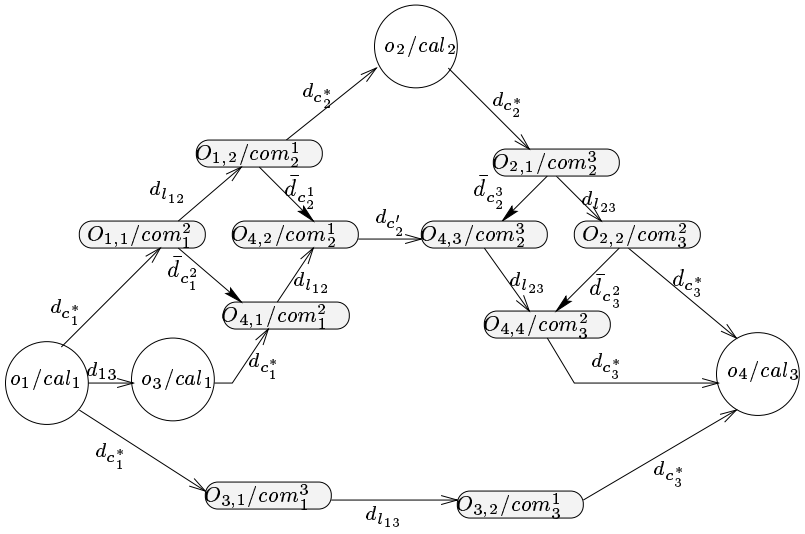


FIG. 7 – *Ordonnancement vue topologique*

3.3.4 Composition des applications

Rappel sur les compositions de relations :
Soit $\mathcal{R}_1 \subseteq A \times B$
Soit $\mathcal{R}_2 \subseteq B \times C$
On définit $(\mathcal{R}_2 \circ \mathcal{R}_1)(a)$ par :
$(\mathcal{R}_2 \circ \mathcal{R}_1)(a) = \{c \in C$ tel que pour au moins un $b \in B, b \in \mathcal{R}_1(a)$ et $c \in \mathcal{R}_2(b)\}$

Simplification des notations :
On pose :

$$\begin{array}{lcl} G_{lm} & = & (G_l, G_m) \\ G_{lm'} & = & (G_l, G'_m) \\ G_{dRm'}(i) & = & (G_{dR}(i), G'_m) \\ G_{Sm'}(i,j) & = & (G_S(i,j), G'_m) \end{array}$$

et donc, en utilisant les nouvelles notations, on a :

$$
\begin{array}{lcl}
\mathcal{R}_{route}(G_{lm}) & = & \{G_{lm'}\} \\
\mathcal{R}_{dis}(G_{lm'}) & = & \{G_{dRm'}(1), \cdots, G_{dRm'}(n)\} \\
\mathcal{R}_{ord}(G_{dRm'}(1)) & = & \{G_{Sm'}(1,1), \cdots, G_{Sm'}(1,N_1)\} \\
\quad \vdots & & \\
\mathcal{R}_{ord}(G_{dRm'}(i)) & = & \{G_{Sm'}(i,1), \cdots, G_{Sm'}(i,N_i)\} \\
\quad \vdots & & \\
\mathcal{R}_{ord}(G_{dRm'}(n)) & = & \{G_{Sm'}(n,1), \cdots, G_{Sm'}(n,N_n)\}
\end{array}
$$

et donc l'implantation est la composition des 3 relations précédentes ce qui donne pour un couple G_{lm} donné :

$$
\begin{array}{|l|} \hline
(\mathcal{R}_{ord} \circ \mathcal{R}_{dis} \circ \mathcal{R}_{route})(G_{lm}) = \{G_{Sm'}(1,1), \cdots, G_{Sm'}(1,N_1), \cdots, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad G_{Sm'}(i,1)\ , \cdots, G_{Sm'}(i,N_i), \cdots, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad G_{Sm'}(n,1), \cdots, G_{Sm'}(n,N_n)\} \\ \hline
\end{array}
$$

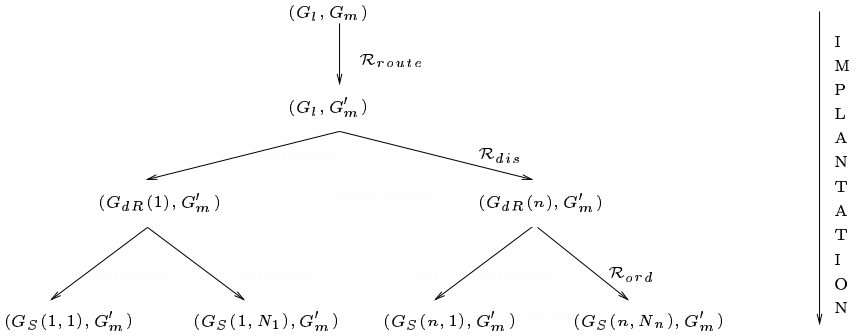


FIG. 8 – *Implantation*

4 Architecture matérielle cible

4.1 Processeur TMS320C40

L'élément de base des machines parallèles (multi-processeurs) que nous considérons est le processeur de traitement du signal (DSP : Digital Signal Processor)TMS320C40 [6] produit par TEXAS INSTRUMENTS. Il comporte six ports de communication (liaisons physiques de communication octet-série bi-directionnelles) ainsi que six canaux de DMA programmables indépendamment les uns des autres. La version utilisée est le TMS320C40 cadencé à 40 MHz. C'est un processeur d'architecture 32-bit comportant une Unité Centrale de Programmation (CPU) autorisant deux accès mémoire par cycle d'instruction et une unité de transfert

mémoire multiplexée (DMA) pour chacun des six ports. Une fois paramétré et lancé par le CPU, chaque canal de DMA peut effectuer ses transferts sans nécessiter aucune ressource du CPU. Ce dernier peut donc effectuer des calculs en parallèle avec les communications.

4.2 Caractéristiques des communications sur le TMS320C40

Les architectures multi-processeurs permettent d'augmenter la puissance de calcul par rapport à un mono-processeur. Elles nécessitent de découper le programme de départ en autant de programmes que de processeurs et impliquent donc des communications entre processeurs. Chaque processeur doit communiquer des résultats de son travail à un autre processeur, ce transfert pouvant s'effectuer en parallèle avec une séquence d'instructions sur le CPU. Les communications inter-processeurs sont critiques dans la conception de système multi-processeurs.

Les sytèmes multi-processeurs à hautes performances nécessitent des transferts de données rapides entre processeurs. Pour assurer ces transferts rapides de données, le TMS320C40 propose ce qui suit :

- une mémoire partagée ;
- des ports de communication à grands débits.

Nous ne nous intéresserons ici qu'aux ports de communication.

4.3 Ports de communications

Le TMS320C40 possède six ports de communications bi-directionnels pour des transferts rapides entre processeurs (5-Megamots par seconde à 40 MHz).

Chaque port de communication contient les composants suivants :

- un "input FIFO channel" qui fournit un tampon d'entrées de 8 niveaux de 32 bits de large ;
- un "output FIFO channel" qui fournit un tampon d'entrées de 8 niveaux de 32 bits de large ;
- une unité d'arbitrage du port (PAU) qui gère l'arbitrage associé au déplacement de données entre un TMS320C40 et un élément externe au travers du bus de données du port de communication ;
- un registre de contrôle du port de communication (CPCR) qui autorise le contrôle des fonctions du port de communication et des opérations de transferts de données entre un TMS320C40 et un élément externe via le bus de données du port de communication.

Se référer à [6, chapitre 8] pour de plus amples informations sur les ports de communications du TMS320C40.

4.4 Coprocesseur de DMA (Direct Memory Access)

Le DMA est un coprocesseur programmable permettant des transferts de données sans aucune intervention du CPU. Le coprocesseur de DMA [6, chapitre 9] comporte six canaux qui permettent des transferts depuis et vers n'importe quelle adresse mémoire du processeur. Il peut automatiquement réinitialiser ses registres par l'intermédiaire de données chargées en mémoire, permettant au DMA d'opérer continuellement sans aucune intervention du CPU. Le DMA peut gérer des tampons circulaires en mémoire et exécuter des adressages linéaires ou en bit inversé (bit-reversed). Il possède également un mode spécial (split mode) permettant de dédoubler chaque canal et obtenir ainsi 12 canaux de DMA pour des transferts entre port de communication et mémoire. Le DMA permet aussi la synchronisation des transferts à l'aide d'interruptions externes ou internes.

5 Algorithme de détection de contours

Une image est un signal à deux dimensions représenté par un tableau bidimensionnel composé de pixels. Chaque pixel possède une localisation (coordonnées (x, y)) et une valeur de luminosité, donnée brute issue d'un capteur après conversion analogique/numérique.

Un contour dans une image représente une variation de la fonction intensité de pixels voisins. C'est une courbe siège de transitions locales de la fonction d'intensité lumineuse de l'image, notée $I(x, y)$.

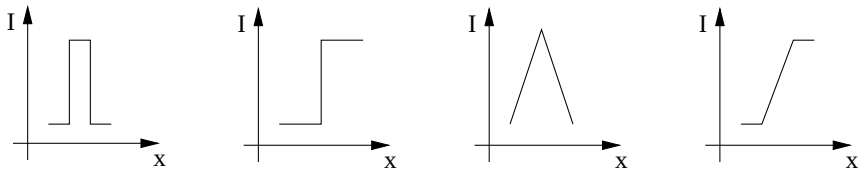
Un contour peut donc signaler :

- une transition brusque entre deux régions de l'image d'intensités moyennes très dissemblables
- une transition graduelle avec un extremum d'intensité séparant deux régions d'intensités variables
- un ensemble de pixels de même intensité séparant une partie de l'image en deux régions de même intensité
- une brusque transition du gradient local de l'image (ligne de crête)

Un point de contour est un pixel qui est identifié comme appartenant à un contour de l'image et qui donc, vérifie des propriétés de discontinuité.

Voici un schéma qui représente un échantillon des contours les plus fréquemment rencontrés. La représentation se fait sous forme de profils mono-dimensionnels qui sont des coupes dans la direction perpendiculaire au contour de la fonction intensité.

L'algorithme de détection de points de contours peut se décomposer en 4 sous-algorithmes enchaînés : un lissage, un gradient, une binarisation et un affinage [7].



(a) Divers types de profils de contours

5.1 Lissage

La détection de contours proprement dite est plus efficace sur une image ayant déjà subi un traitement de lissage.

En effet, une image brute contient en principe une composante de bruit importante qui peut pénaliser la détection directe des éléments de contour. Il s'agit donc d'éliminer une partie de la composante de bruit qui est en fait constituée de hautes fréquences spatiales, sans pour autant altérer l'information contenue dans l'image. On applique donc un filtre passe-bas sur l'image initiale afin d'atténuer le bruit.

Pour cela, nous avons choisi de réaliser un lissage linéaire, et plus précisément une moyenne. L'inconvénient de ce type de lissage est d'atténuer les contours des images, puisque les contours induisent eux-mêmes des hautes fréquences spatiales.

5.1.1 Principe du lissage

Soit $I[m,n]$, la matrice de m lignes et n colonnes, représentant l'image.

Soit (x,y), le point de l'image, en cours de traitement.

Soit $V[p,p]$, le voisinage d'un point de cette image, p étant la longueur et la largeur du voisinage.

Soit $M[p,p]$, le masque qui va être appliqué séquentiellement sur tous les points de l'image.

Soit $I_l[m,n]$, la matrice contenant les points lissés.

Le lissage consiste à effectuer une convolution de l'image par un masque $p \times p$ sur l'entourage de chacun des pixels pris de manière séquentielle.

On a donc :

$I_l(x,y) = M * V$ avec $V(i,j) = I(x+i, y+j)$ et $1 \leq i,j \leq p$; $M(i,j) = \frac{1}{p^2}$

$I_l(x,y) = \sum_{i=1}^{p} \sum_{j=1}^{p} M(i,j)V(i,j)$

$I_l(x,y) = \frac{1}{p^2} \sum_{i=1}^{p} \sum_{j=1}^{p} V(i,j) = \overline{V}(i,j)$ par définition de la moyenne empirique.

$I_l(x,y)$ est donc bien un point dont l'intensité est égale à la moyenne des points appartenant à son voisinage.

Le masque choisi pour représenter l'opérateur de moyennne sur un voisinage 3×3 est le suivant :

$$M = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

5.1.2 Algorithme

Boucle sur les pixels de l'image
Pour $x = 1, m$, pour $y = 1, n$
  Initialisation du nouveau point considéré
  $I_l(x,y) = 0$
  Boucle sur le voisinage du point considéré
  Pour $i = 1, p$, pour $j = 1, p$
    $I_l(x,y) = I_l(x,y) + M(i,j) * I(x+i)(y+j)$
  Fin pour i, fin pour j
Fin pour x, fin pour y

5.2 Gradient

Les transitions en niveau de gris de l'image sont détectables grâce au gradient. En effet, une zone de transition dans l'image correspond à un extremum de la norme du gradient. Pour mettre en évidence cette zone de transition ou encore ces contours, nous avons choisi d'effectuer un seuillage de cette norme, ce qui correspond à la binarisation qui sera définie au paragraphe suivant.

5.2.1 Principe du gradient

Soit $f(x,y)$, une fonction de deux variables, bornée et dérivable en tout point.

Le gradient de f est défini par :

$\nabla f = \begin{vmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{vmatrix}$

Le vecteur gradient en un point M de coordonnées (x,y), est normal à la courbe de niveau $f(x,y) = constante$ qui passe par M. En effet, soit $\vec{r} = x\vec{i} + y\vec{j}$, le rayon vecteur d'un point quelconque de cette courbe.

Alors : $d\vec{r} = dx\vec{i} + dy\vec{j}$ est tangent en M à cette courbe.

Et $df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy = 0$ or $df = \vec{\nabla} f.dr = 0$

$\Longrightarrow \vec{\nabla} f \perp d\vec{r}$

$\Longrightarrow$ *Le gradient est orthogonal à la direction du contour* .

$\|\vec{\nabla} f\| = \sqrt{|\frac{\partial f}{\partial x}|^2 + |\frac{\partial f}{\partial y}|^2}$

$\|\vec{\nabla} f\|$ sera d'autant plus élevée que la fonction f sera variable d'un point à un autre. Autrement dit, un point de contour siège d'une zone de transition sera reconnaissable par la norme de son gradient qui sera supérieure à un certain seuil qu'il faudra définir.

5.2.2 Gradient directionnel

Pour un contour d'orientation θ, le vecteur unitaire normal au contour et donc parallèle au gradient directionnel, a pour expression au point (x, y) :
$\vec{n} = (cos\Phi, sin\Phi)$ et $\theta = \frac{\pi}{2} + \Phi \quad [\pi]$
La dérivée directionnelle de $f(x, y)$ suivant $\vec{n}$ s'écrit :

$$\boxed{\frac{\partial f}{\partial n} = \vec{\nabla} f(x, y).\vec{n} = cos\Phi \frac{\partial f}{\partial x} + sin\Phi \frac{\partial f}{\partial y}}$$

Soit I, l'image numérique résultante de l'échantillonnage de la fonction scalaire f.
Approximation du gradient au point (x, y) : (formule de Taylor)

$$\begin{cases} \frac{\partial I}{\partial x}(x, y) = I(x+1, y) - I(x-1, y) \\ \frac{\partial I}{\partial y}(x, y) = I(x, y+1) - I(x, y-1) \end{cases}$$

Gradient au point (x, y) **dans la direction** $\vec{\imath}$ (pour détecter un contour vertical)
$\Phi = 0$ et $\theta = \frac{\pi}{2}$ et $\vec{n} = \vec{\imath}$
$\Longrightarrow \frac{\partial I}{\partial n} = \frac{\partial I}{\partial x}$
ce qui se représente dans la base $(I(x-1, y), I(x, y), I(x+1, y))$ par le masque élémentaire $(-1, 0, 1)$

Gradient au point (x, y) **dans la direction** $\vec{\jmath}$ (pour détecter un contour horizontal)
$\Phi = \frac{\pi}{2}$ et $\theta = 0$ et $\vec{n} = \vec{\jmath}$
$\Longrightarrow \frac{\partial I}{\partial n} = \frac{\partial I}{\partial y}$ ce qui se représente dans la base $\begin{pmatrix} I(x, y-1) \\ I(x, y) \\ I(x, y+1) \end{pmatrix}$ par le masque élémentaire $\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$.

Gradient au point (x, y) **dans la direction** $\vec{u} = \vec{\imath} + \vec{\jmath}$ (pour détecter un contour à 45 degrés par rapport à l'horizontale)
$\Phi = \frac{3\pi}{4}$ et $\theta = \frac{\pi}{4}$ et $\vec{n} = \vec{\imath} + \vec{\jmath} \quad \Longrightarrow \frac{\partial I}{\partial n} = \frac{\partial I}{\partial x} + \frac{\partial I}{\partial y}$
ce qui se représente dans la base $\begin{pmatrix} & I(x, y-1) & \\ I(x-1, y) & I(x, y) & I(x+1, y) \\ & I(x, y+1) & \end{pmatrix}$
par le masque élémentaire $\begin{pmatrix} & -1 & \\ -1 & 0 & 1 \\ & 1 & \end{pmatrix}$.

Gradient au point (x, y) **dans la direction** $\vec{u} = \vec{\imath} - \vec{\jmath}$ (pour détecter un contour à 135 degrés par rapport à l'horizontale)

$\Phi = \frac{\pi}{4}$ et $\theta = \frac{3\pi}{4}$ et $\vec{n} = \vec{\imath} - \vec{\jmath}$ $\quad \Longrightarrow \frac{\partial I}{\partial n} = \frac{\partial I}{\partial x} - \frac{\partial I}{\partial y}$

ce qui se représente dans la base $\begin{pmatrix} & I(x,y-1) & \\ I(x-1,y) & I(x,y) & I(x+1,y) \\ & I(x,y+1) & \end{pmatrix}$

par le masque élémentaire $\begin{pmatrix} & -1 & \\ 1 & 0 & -1 \\ & 1 & \end{pmatrix}$.

5.2.3 Allure des masques

Pour éviter de détecter les points, siège de transitions locales, mais isolés, il est nécessaire de chercher les points qui ont dans leur voisinage des points vérifiant les mêmes propriétés de discontinuité. Cela revient à prendre des masques plus grands que ceux précédemment définis. Nous avons choisi de prendre des masques 3×3. Ces nouveaux masques sont construits à partir des masques élémentaires de la manière suivante : le masque élémentaire est dupliqué dans la direction du contour en sachant que à un point du contour correspond un 0 du masque et que les autres coefficients du masque doivent être de signe constant de part et d'autre du contour représenté par des zéros puisque les lignes de niveau sont sensiblement égales de part et d'autre du contour.

Les masques obtenus correspondent à l'opérateur de Kirsh à 4 directions.

contour vertical	contour horizontal	contour oblique	contour oblique
$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{pmatrix}$

5.2.4 Algorithme

Soit $I_l[m,n]$, la matrice représentant l'image.

Soient M_1, M_2, M_3, M_4, les masques 3×3 approximant les gradients dans les 4 directions.

Soient I_1, I_2, I_3, I_4, les images $m \times n$ obtenues après convolution de l'image originale $I[m,n]$ avec les 4 masques.

Alors, l'image résultant de l'approximation de la norme du gradient est donnée au point (x,y) par :

$I_{norme}(x,y) = \sqrt{I_1^2(x,y) + I_2^2(x,y) + I_3^2(x,y) + I_4^2(x,y)}$

5.3 Binarisation

Le calcul de la norme du gradient a permis de localiser l'emplacement des points de contours présents dans l'image. La binarisation permet de les mettre en évidence. Au-delà d'un certain seuil, l'intensité de la norme correspond à un point de contour, et en-deçà de ce seuil, l'intensité de la norme indique un point appartenant au fond de l'image.

Soit I_{norme}, l'image représentant l'approximation de la norme du gradient.

Soit I_{bin}, l'image des points de contour.

Soit S, le seuil choisi.
Si $I_{norme}(x,y) < S$ alors $I_{bin}(x,y) = 0$ ((x,y) appartient au fond de l'image)
Sinon, $I_{bin}(x,y) = 1$ ((x,y) est un point de contour)

5.4 Affinage

Après une détection de points de contours, l'image obtenue est une image binarisée des points de contours. Mais, il se peut aussi, et ce fut le cas de notre application, que le contour ait plusieurs pixels d'épaisseur. Il faut donc appliquer une procédure d'affinage, utilisant l'opérateur morphologique érosion, afin de réduire l'épaisseur des contours.

5.4.1 Définition de l'opérateur

Soit I_{bin}, l'image binarisée des points de contour.
Alors : $I_{bin} = X \cup X^c$ où X est l'ensemble des points de contour et X^c est l'ensemble des points constituant le fond de l'image.
Soit B, l'élément structurant, i.e. un autre ensemble, qui va parcourir X grâce à des translations par le vecteur u.
L'érosion est notée :
$X \ominus \check{B} = \{u,\ B_u \subset X\}$
où $X_b = \{b + u,\ u \in X\ \}$ est le translaté de X par b
Autrement dit, l'érodé morphologique de X (donc l'érodé des points de contours), par l'élément structurant B est l'ensemble des pixels u de X pour lesquels B centré en u est inclus dans X.

5.4.2 Principe de l'affinage

Ce traitement sera effectué sur une image binaire et restituera une image binaire contenant les contours affinés. Ainsi, un pixel appartenant à un contour ne sera conservé que si tous les pixels de son voisinage sont des pixels de contour ; en supposant qu'un pixel de contour soit codé par 1 et qu'un pixel du fond de l'image soit codé par 0, un pixel appartiendra au contour affiné si il n'y a aucun pixel à 0 dans son voisinage. Celà revient à faire un ET logique entre le masque et le voisinage du point considéré. On compare le résultat du ET logique avec le masque et s'ils sont identiques, le point de contour est conservé.
Le masque choisi est le suivant:

$$M = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

5.4.3 Algorithme

On compare le voisinage de tout point $\neq 0$ et n'appartenant pas aux bords avec le masque.
Soit $M[p_1, p_2]$, la matrice représentant le masque.
Soit $I[p_1, p_2]$, la matrice représentant le voisinage du point considéré.

Soit $R[p_1, p_2]$, la matrice résultat du ET logique. Elle est construite comme suit : pour tout point $(i, j) \neq 0$ de l'image de départ :
Si $M(i, j) \wedge I(i, j) = vrai$ alors $R(i, j) = vrai = 1$
Sinon $R(i, j) = faux = 0$

Enfin, on compare M et R.
Si $M(i, j) = R(i, j)$ $\forall i$, $\forall j$, $1 \leq i \leq p_1$, $1 \leq j \leq p_2$ alors $M(p_1/2, p_2/2) = 1$
c'est-à-dire que le point considéré reste à 1.
Sinon, il est mis à zéro, ce qui signifie qu'une partie de l'objet contenue dans l'image a été affinée.

Cet ensemble de traitements successifs réalise donc la détection de points de contours d'une image. Chacun de ces traitements a été spécifié avec le langage SIGNAL.

5.5 Gestion des bords d'une image

Chacun des sous-algorithmes de l'algorithme complet de l'application détection de contours, sera appliqué sur une matrice de taille $m \times n$ et restituera une matrice $m \times n$, ceci dans un souci d'homogénéité et d'adaptabilité des processus les uns par rapport aux autres.

L'application de la moyenne ainsi que celle de la norme du gradient impliquent de passer des masques sur l'image. Les masques considérés sont de taille $p \times p$ ($p = 3$ pour notre application) donc le problème du traitement des bords de l'image se pose. En effet, un bord de l'image d'une épaisseur de $(p - 2)$ pixel ne sera pas traité. Ces traitements nécessitent donc une action préalable : on va agrandir l'image initiale à la taille $(m + 2) \times (n + 2)$ pour qu'au passage des masques, les pixels de l'image soient bien considérés comme des pixels courants.

Le choix initial était de créer un tour de pixels nuls autour de l'image à traiter mais les images obtenues après une moyenne par exemple comportait des effets de bord non négligeables. Nous avons donc changé de méthode en dupliquant les bords selon un effet miroir.

6 Spécification de l'algorithme avec SIGNAL

6.1 Langages synchrones

Les langages classiques de spécification (C, Fortran, ...) sont difficilement utilisables pour spécifier des applications temps réel car, d'une part ils sont mal adaptés à l'exploitation du parallélisme potentiel, et d'autre part leur sémantique ne possède pas la notion de "temps". A l'inverse, *les langages synchrones* [8] permettent l'expression du parallélisme présent dans un algorithme et la modélisation du temps.

Aussi, nous allons présenter un tel langage développé à l'INRIA de Rennes.

6.2 Présentation de SIGNAL

SIGNAL [9] [10] est utilisé dans l'application de la méthodologie Adéquation Algorithme Architecture pour spécifier et vérifier des algorithmes d'application hors contraintes matérielles. C'est un langage *flot de données, synchrone*, permettant de décrire des relations entre des suites d'événements valués, les *signaux*. Il y a trois types de relations :

- une relation de *précédence* entre les événements d'un même signal, ce qui permet de les compter et donc de dater leurs valeurs selon un temps logique ;
- une relation de *simultanéité* entre événements de signaux différents. On peut toujours savoir s'ils coïncident ou s'ils sont distincts ;
- des relations de *dépendances opératoires* qui définissent des signaux de sortie (réactions), résultats d'opérations faites sur des signaux d'entrée(stimuli).

L'hypothèse synchrone signifie qu'on ne prend pas en considération les caractéristiques matérielles (les durées des actions internes sont négligées donc les propriétés d'un programme ne sont plus liées aux vitesses d'exécution, ni aux débits de communication). On suppose alors que les sorties produites par une dépendance opératoire sont simultanées avec les entrées. Deux signaux sont synchrones si leurs événements sont deux à deux simultanés (présents au même instant) ; ceci définit alors une relation d'équivalence. La classe d'équivalence d'un signal X est appelée son horloge, P(X), qui définit ses instants de présence (ou d'absence) relativement à d'autres signaux. On définit aussi T(b) l'horloge des valeurs vraies d'un signal booléen b qui sert au conditionnement, on a alors la relation : $T(b) \subseteq P(b)$. L'ensemble des horloges d'un programme SIGNAL, indiquant la présence et l'absence des événements des signaux les uns par rapport aux autres, définit son "temps logique".

Pour décrire des relations opératoires, on dispose de quatre *processus élémentaires* et d'une *instruction de composition* de processus élémentaires ("|"). Cette dernière instruction établit des connexions par identité de nom (flots de données), entre entrée et sortie de processus élémentaires.

En SIGNAL, un programme est un processus obtenu par composition de processus élémentaires et/ou de processus.

Les 4 processus élémentaires sont :

- La *fonction immédiate*
  Chaque valeur de l'événement du signal de sortie $f(x)$ est le résultat de la fonction f appliquée à la valeur de l'événement correspondant du signal d'entrée x. Le signal de sortie dépend uniquement du signal d'entrée dans l'instant logique considéré. Les signaux d'entrée et de sortie ont la même horloge.
  $X := A + B$ $\quad$ X somme de A et B $\quad$ $P(X) = P(A) = P(B)$
  Diagramme temporel logique correspondant :

$A=$	1	2	3	4	5
$B=$	1	2	3	4	5
$X := A + B$	2	4	6	8	10

– Le *retard*
Il matérialise les valeurs passées d'un signal. C'est une mémorisation qui nécessite l'initialisation du signal de sortie pour le traitement du premier instant logique. Les signaux d'entrée et de sortie ont la même horloge mais le signal de sortie ne dépend pas du signal d'entrée.

$Z := X\$M$ Z retardé de M échantillons par rapport à X $P(Z) = P(X)$
Diagramme temporel logique correspondant :

$X=$	1	2	3	4	5	
$ZX := X\$1$	0	1	2	3	4	ZX initialisé à 0

– Le *sous-échantillonage*
Il permet d'extraire certains événements d'un signal. La sortie prend la valeur de l'entrée quand l'entrée (de type quelconque) est présente et que l'autre entrée de conditionnement (de type booléen) est présente et vraie. Le signal de sortie dépend du signal d'entrée uniquement si le signal de conditionnement est présent et vrai ; l'entrée est sous-échantillonée par les valeurs vraies du booléen de conditionnement. L'horloge du signal de sortie sera définie par l'intersection de l'horloge du signal d'entrée avec l'horloge vraie du signal de conditionnement.

$Y := X$ *when* B X sous échantilloné par B vrai $P(Y) = P(X) \cap T(B)$
Diagramme temporel logique correspondant :

$X =$	1	2	3	4	$\perp$	5	6	$\perp$	7	8
$B =$	F	T	$\perp$	T	T	F	T	F	$\perp$	T
$Y =$	$\perp$	2	$\perp$	4	$\perp$	$\perp$	6	$\perp$	$\perp$	8

– Le *mélange prioritaire*
Il permet de mélanger des signaux de même type, ayant des horloges différentes. La sortie prend la valeur de celle des deux entrées qui est présente, si les deux entrées sont présentes, c'est toujours la même entrée qui est prise de manière prioritaire (membre de gauche du $default$). Le signal de sortie dépend donc du signal x0 lorsque x0 est présent ou du signal x1 lorsque x1 est présent et x0 absent. L'horloge du signal de sortie sera définie par l'union des horloges des signaux d'entrée.

$X := X0$ *default* $X1$ mélange de $X0$ et $X1$ $P(X) = P(X0) \cup P(X1)$
Diagramme temporel correspondant :

$X0 =$	1	3	$\perp$	5	6	$\perp$	8	9
$X1 =$	$\perp$	2	4	2	$\perp$	$\perp$	9	6
$X =$	1	3	4	5	6	$\perp$	8	9

6.3 Vérification temporelle logique

Le compilateur SIGNAL effectue une vérification temporelle du programme, indépendamment de l'architecture sur laquelle il sera implanté. La cohérence de l'ordre logique des événements des signaux est vérifiée grâce aux horloges des signaux. A chaque programme est associé un système d'équations d'horloges qui doit admettre une solution unique pour être correct. Si certains signaux ont une horloge indéterminée, il faut les synchroniser avec des signaux d'horloge connue. L'instruction à utiliser est : SYNCHRO $\{X, Y\}$ où Y, signal d'horloge indéterminée, prend l'horloge de X déterminée.

En fonction des options de compilation, deux fichiers peuvent être générés :

- un fichier qui contient le programme source SIGNAL et éventuellement les erreurs de syntaxe (nom du programme_LIS) ;
- un fichier issu du calcul d'horloge qui contient la hiérarchie des horloges et éventuellement les erreurs de synchronisation des signaux (nom du programme_TRA).

6.4 Génération de code C

A partir du programme source SIGNAL le compilateur génère, avec l'option -c, à l'aide de la commande :

```
sig -c <nom du fichier>.sig
```

trois fichiers C ayant comme nom le nom du programme source SIGNAL, suffixés par _E.c, _M.c, _S.c (cf. programmes générés en C par le compilateur SIGNAL en annexe B.3). Après compilation et link de ces fichiers, l'exécutable obtenu permet de simuler l'application sur une station de travail (mono-processeur) **sans tenir compte des contraintes temps réel**.

Le fichier _E.c

Ce sont des fonctions qui s'occupent de gérer les signaux d'Entrée-Sorties déclarés dans le programme SIGNAL. Chaque signal est simulé par un fichier séquentiel de données représentant les valeurs prises. Ce programme C va lire dans des fichiers identifiés par le nom du signal précédé de R (READ) et écrire dans des fichiers dont le nom est précédé de W (WRITE).

Dans le cadre de notre application de traitement d'images, nous avons un fichier d'entrée contenant la séquence des images à traiter et un fichier de sortie contenant la séquence des images résultats. A chaque instant logique, une image correspondant à un événement d'entrée est traitée. Chaque image étant codée au format PGM, nous devons modifier ces fichiers de la manière suivante (cf. programme de gestion des entêtes en annexe C) :

- dans le fichier d'entrée précédé de R chaque image doit être débarrassée de son entête pour que les traitements puissent la considérer comme une matrice 2D ;

– le fichier précédé de W contient les images résultats des traitements effectués par le programme SIGNAL. Pour les visualiser, il faut leur rajouter une entête de manière à ce que le logiciel de visualisation sache que c'est une image.

Il faut noter que dans un instant logique une image est traitée par le programme SIGNAL pixel à pixel.

Le fichier _S.c

Il contient la procédure C générée par le compilateur SIGNAL réalisant les calculs et le contrôle sur les signaux.

Le fichier _M.c

C'est la fonction principale (main) qui appelle les autres fonctions C générées par le compilateur SIGNAL.

6.5 Etude de la granulation et gestion des données

La première étape a consisté à écrire, en SIGNAL, un programme réalisant la moyenne, calculant la norme du gradient, réalisant un seuillage puis une érosion d'une image. Il a été écrit au niveau le plus fin de granulation en explicitant tous les calculs. On exhibe ainsi à la fois du parallélisme de données et du parallélisme pipe-line. Par exemple, la moyenne a été décrite comme une somme de produits. Le graphe flot de données correspondant à la moyenne comprend des nœuds *somme* et *produit* interconnectés.

Une deuxième étape a consisté à changer la granulation pour ne faire apparaître que l'enchaînement des fonctionnalités principales. Le graphe flot de données correspondant ne comprend que les quatre nœuds : moyenne, gradient, seuillage, érosion. On exhibe ainsi uniquement du parallélisme pipe-line potentiel (cf. figure 12 en annexe D et programme SIGNAL en annexe A).

Après avoir testé l'algorithme sur des images réelles de taille (160×100), nous avons décidé, dans une nouvelle étape, d'améliorer le parallélisme de notre algorithme en partageant l'image de départ en 4 bandes horizontales appelées "imagettes". On fait alors apparaître un nouveau niveau de granulation où chaque nœud utilisé dans l'étape précédente est découpé en quatre afin de mettre en évidence du parallélisme de données. Par exemple, on applique en parallèle quatre fois l'opération moyenne à chaque imagette. On a ici du parallélisme de données et du parallélisme pipe-line à un niveau moins fin que lors de la première étape. (cf. source SIGNAL en annexe B.1, procédures C en annexe B.2 et graphes SynDEx figure 13 annexe D). Chaque imagette a besoin d'informations sur des pixels appartenant aux autres imagettes. Voici la solution que nous avons choisie pour ne pas perdre d'information entre les imagettes :

– les bords latéraux extérieurs des imagettes sont obtenus par duplication des bords intérieurs (effet miroir) ;

- le bord extérieur supérieur (resp. inférieur) de la bande supérieure (resp. inférieure) est obtenu par duplication du bord supérieur (resp. inférieur) de la bande supérieure (resp. inférieure);
- le bord extérieur supérieur d'une imagette intermédiaire est la ligne inférieure intérieure de l'imagette du dessus ;
- le bord extérieur inférieur d'une imagette intermédiaire est la ligne supérieure intérieure de l'imagette du dessous.

Ce niveau de granulation intermédiaire a permis d'exploiter le parallélisme effectif d'une architecture à quatre processeurs à partir du parallélisme potentiel de l'algorithme.

7 Implantation de l'algorithme avec SynDEx

7.1 Environnement logiciel

SynDEx [11] [12] est un environnement logiciel graphique interactif de développement pour applications temps-réel de commande, de traitement du signal et des images, supportant la méthode d'Adéquation Algorithme Architecture. Il permet, à partir de la spécification de l'algorithme et de l'architecture de conduire à une implantation efficace sur machine multi-processeurs, c'est-à-dire respectant des contraintes temps réel et minimisant les ressources matérielles (nombre de processeurs et de liaisons de communication). Enfin, il génère automatiquement des exécutifs optimisés gérant entre autre un système de communications sans inter-blocage. Ceci décharge l'utilisateur de la programmation bas niveau et supprime les tests en multi-processeur, dans le cas ou l'on suppose le matériel sans panne.

SynDEx permet de spécifier l'algorithme et l'architecture matérielle sur laquelle il doit s'exécuter, en saisissant à la souris les graphes logiciels (algorithmes) et matériels (architectures) dans la fenêtre "édition", voir figure 9. Le graphe logiciel peut aussi être importé à partir d'un fichier produit lors de la compilation d'un programme SIGNAL avec l'option de génération de code SynDEx au lieu de C [13].

L'implantation de l'algorithme sur l'architecture consiste à distribuer les opérations sur les processeurs (allocation spatiale, utilisation du parallélisme disponible) et à ordonnancer sur chaque processeur les opérations qu'il doit exécuter (allocation temporelle, réduction du parallélisme potentiel). Des heuristiques permettent de réaliser une implantation efficace (Adéquation Algorithme Architecture) selon divers critères. Nous nous intéresserons ici uniquement à l'optimisation du temps de réponse [2].

Après exécution de l'heuristique d'optimisation du temps de réponse (latence), un schéma prévisionnel (diagramme temporel) d'exécution temps-réel de l'algorithme sur le multi-processeur peut être visualisé dans la fenêtre "schedule", voir figure 9. On peut alors observer les communications inter-processeurs et les opérations ordonnancées sur chaque processeur. Le temps est lu selon une échelle verticale et chaque colonne contient les opérations liées à un processeur. Une communication inter-processeurs est symbolisée par un segment de

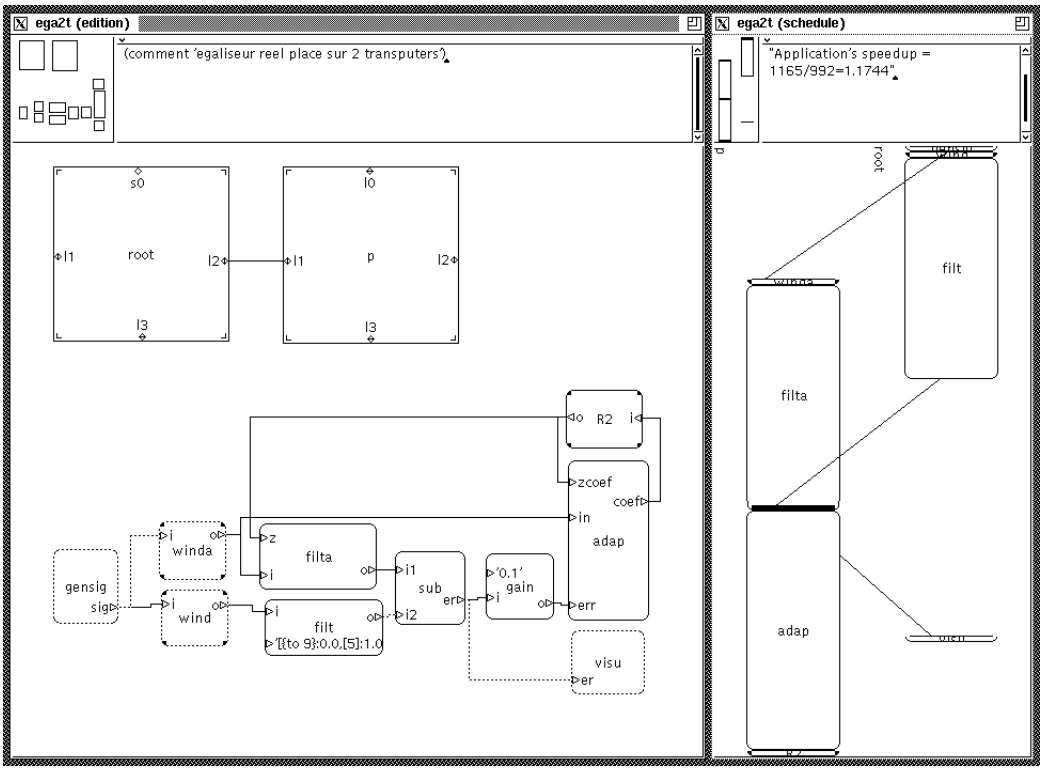


FIG. 9 – *Environnement SynDEx*

droite dont la longueur de la projection sur l'axe du temps est proportionnelle à sa durée. Une opération est représentée par une boîte de hauteur proportionnelle à la durée de l'opération. La durée de chaque boîte est mesurée, au préalable, à l'aide d'une horloge temps réelle ou estimée quand ce n'est pas possible. Ces durées sont fournies lors de la spécification du graphe logiciel. Il est important de noter que la qualité des résultats donnés par l'heuristique d'optimisation dépend de la qualité de la mesure de ces durées.

Enfin, un exécutif [14] est généré automatiquement pour chaque processeur. Il sera de préférence au maximum statique afin de limiter le surcoût qu'il induit. Il évolue au fil des versions de SynDEx vers un exécutif allouant toutes les ressources de façon statique comme dans la version "minimum" de la dernière ligne du tableau 1. Pour cela il faut passer d'un ordonnancement dynamique des calculs et des communications à un ordonnancement statique des calculs et des communications en les séquentialisant, tout en gardant lorsque cela est possible du parallélisme entre calculs et communications. Ceci correspond à la version v4 de SynDEx en cours de développement.

Exemples d'exécutifs	calcul		mémoire		communic.	
	dist	ord	dist	ord	dist	ord
Meiko	dyn	dyn	dyn	dyn	dyn	dyn
CHORUS	sta	dyn	sta	dyn	dyn	dyn
SynDEx v1	sta	dyn	sta	dyn	sta	dyn
SynDEx v2, v3	sta	sta	sta	dyn	sta	dyn
minimum	sta	sta	sta	sta	sta	sta

TAB. 1 – *Différents types d'exécutifs*

Les exécutifs sont construits à partir, d'une part des opérations de calcul et d'entrée-sortie fournies par l'utilisateur, et d'autre part d'un noyau générique fournissant les fonctionnalités de conditionnement, de mémorisation, de synchronisation et de communication.

En version v3 de SynDEx, les exécutifs sont des programmes C. L'allocation mémoire et l'ordonnancement des opérations sont directement supportés par le langage C (variables pour l'allocation mémoire, structures de contrôle pour le séquencement conditionnel ou inconditionnel). Les autres fonctionnalités que supportent l'exécutif sont :

- l'initialisation des mémoires programme des processeurs ;
- le lancement en parallèle des programmes (un par processeur) ;
- le lancement en pseudo-parallèle des processus sur chaque processeur ; (afin de tirer parti du parallélisme interne à chaque processeur composé d'unités pouvant exécuter en parallèle des opérations mais partageant un même séquenceur) ;
- les communications inter-processeur ;
- les entrées-sorties avec l'environnement du multi-processeur ;
- les mesures de performances de l'exécution multi-processeur.

Ces dernières fonctionnalités sont dépendantes du type des processeurs composant l'architecture et sont regroupées dans un fichier `include syndex.h` (déclarations de types, de macros et d'interface de fonctions compilées séparément) et dans une librairie `syndex.lib` (regroupant les fonctions compilées séparément).

L'ensemble de ces fonctionnalités, celles indépendantes et celles dépendantes du matériel, composent le "noyau générique d'exécutifs SynDEx".

Enfin, l'exécutif peut être généré optionnellement avec des instructions de chronométrage pour obtenir les durées d'exécution réelles des opérations. Le chronométrage est réalisé en enregistrant les dates de début et de fin de toutes les opérations de calcul et de communication en utilisant l'horloge temps réel des processeurs. Ces durées mesurées peuvent être utilisées ultérieurement par l'heuristique d'optimisation et pour l'évaluation des performances temps réel.

7.2 Création des graphes logiciels et matériels

Dans l'environnement logiciel SynDEx, on peut avoir plusieurs fenêtres contenant chacune une application SynDEx. Notre application de détection de contours peut se visualiser sous deux formes :

- topologique (vue édition, figure 10) dans laquelle on édite le graphe logiciel et le graphe matériel ;
- temporelle (vue schedule, figure 11) dans laquelle on visualise les prédictions de performances temporelles de l'implantation de l'algorithme sur l'architecture.

Il est possible de copier, couper, coller les objets d'une fenêtre d'édition d'une application, dans une autre fenêtre d'édition d'une autre application.

La fenêtre d'édition comporte trois parties (cf figure 10) :

- une partie zoom (en haut à gauche) présentant une vue simplifiée des architectures matérielles et logicielles ;
- une partie texte (en haut à droite) dans laquelle on saisit les caractéristiques des boîtes de l'algorithme ou des processeurs ;
- une partie graphe (en dessous des 2 précédentes) dans laquelle on crée, on déplace, on détruit les boîtes représentant des opérations dans le cas du graphe logiciel et les processeurs pour le graphe matériel et les connexions représentant les transferts de données du graphe logiciel et les liaisons de communication du graphe matériel.

Pour créer une boîte représentant une opération, il faut fournir :

- le type de la boîte (fonction immédiate, when, default ...) ;
- le nom de la boîte qui sera visible sur la représentation graphique ;
- le nom de la procédure en langage C qui sera appelée lors de l'exécution temps réel ;
- la durée d'exécution de la procédure (éventuellement fixée arbitrairement dans un premier temps) ;
- les déclarations d'entrée/sortie de la boîte.

Voici un exemple de déclaration de boîte représentant une fonction immédiate :

```
(function MOYENNE "calls" moy "dt" 31000 "i/o"
[25,160] integer? i1, [[160] integer?i2?i3, [25,160] integer!o1,
[160] integer!o2!o3 ).
```

Les informations entre guillemets sont des commentaires.

Pour créer une boîte représentant un processeur, il faut fournir :

- les ports d'entrée/sortie de type PPL (Processor to Processor Link) qui sont des unités de communication bidirectionnelle pour liaisons point-à-point ;

- les ports d'entrée/sortie de type PBL (Processor to Bus Link) qui sont des unités de communication pour liaisons multipoints ;
- le port d'entrée/sortie de type PSL (Processor to Server Link) qui représente l'unité de communication avec le calculateur hôte qui initialise et charge le réseau de processeurs puis gère les entrées/sorties fichiers.

Voici un exemple de déclaration de processeur TMS320C40 :

```
(processor C1 PSL lo, PPL l1 l2 l3 l4 l5)
```

7.3 Adéquation

Dans un premier temps, il faut contraindre les nœuds d'entrée/sortie qui réalisent l'interface avec l'environnement simulé par des fichiers séquentiels, à s'exécuter sur le processeur qui possède l'unique port PSL. Ce port PSL effectue les entrées/sorties fichier sur le disque du calculateur hôte qui n'est pas actuellement représenté dans SynDEx. En lançant l'adéquation, qui exécute l'heuristique de distribution et d'ordonnancement cherchant à minimiser la latence, les autres nœuds non contraints, ainsi que les communications, vont être distribués et ordonnancés sur les processeurs et les liaisons de communication. Sur le graphe de l'algorithme de détection de contours de la figure 10, on peut voir les nœuds (en noir) contraints par l'utilisateur (Ee et Se sont forcés à être exécutés sur le processeur qui contient la PSL) et par l'heuristique, à s'exécuter sur le processeur C1.

Selon la valeur de l'accélération effective délivée par SynDEx, on peut remettre en cause les résultats donnés par l'heuristique en imposant que certains nœuds soient exécutés sur des processeurs particuliers, puis exécuter de nouveau l'adéquation.

C'est donc un processus itératif qui se met en place puisque l'on peut continuer à forcer des nœuds en vérifiant que cela améliore l'accélération.

Le "speed-up effectif" représente l'accélération effective par rapport à une exécution monoprocesseur. Il est calculé en faisant le rapport entre la somme des durées d'exécution des différentes opérations en mode monoprocesseur et la somme des durées calculées lors de l'exécution sur l'architecture multiprocesseurs donnée.

Le "speed-up maximum" représente l'accélération maximum obtenue dans le cas idéal où les communications ne coûteraient rien. Il est calculé en faisant le rapport entre la somme des durées d'exécution des différentes opérations en mode monoprocesseur et le chemin critique du graphe logiciel valué par les durées des opérations. En prenant l'entier directement supérieur au "speed-up maximum" délivré par SynDEx, on obtient le nombre de processeurs, N, à partir duquel il est vain de rajouter des processeurs qui seront inutilisés.

7.4 Diagramme temporel

Après exécution de l'heuristique, SynDEx produit un diagramme temporel qui permet de visualiser la répartition (distribution) des opérations sur chaque processeur, l'ordonnancement des opérations et des liaisons inter-processeurs. La figure 11 représente le diagramme temporel correspondant à l'algorithme de détection de contours implanté sur 4 processeurs

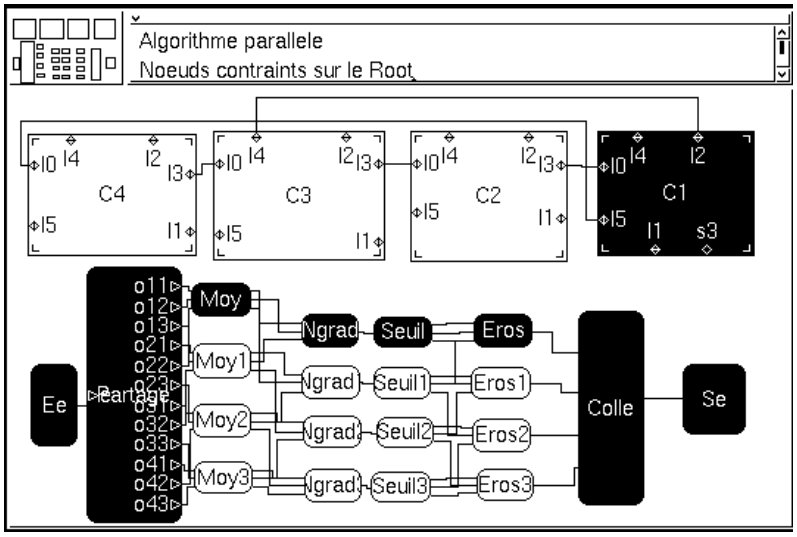


FIG. 10 – *Edition de l'algorithme parallélisé appliqué sur 4 C40 : les opérations en noir sont contraintes à s'exécuter sur le processeur C1*

C40. Sur la partie gauche on peut remarquer que, d'une part la durée des entrée-sorties est grande relativement aux durées de calcul car on accède ici à des fichiers, et d'autre part que c'est l'opération de calcul du gradient qui est l'étape de traitement la plus longue. La partie droite présente un zoom du diagramme temporel.

7.5 Sauvegarde des applications

SynDEx permet de sauvegarder une application dans un fichier d'extension .syn. Ce fichier contient une représentation ASCII des graphes créés et de l'adéquation : les déclarations des boîtes et de leurs connexions, la distribution et l'ordonnancement réalisées par l'utilisateur et l'heuristique et aussi les informations décrivant la position des boîtes dans la fenêtre d'édition.

7.6 Génération des exécutifs

Après avoir réalisé l'adéquation, on peut demander à générer les exécutifs. SynDEx produit :

- un fichier include `test.h` contenant des définitions de constantes symboliques d'informations de routage,

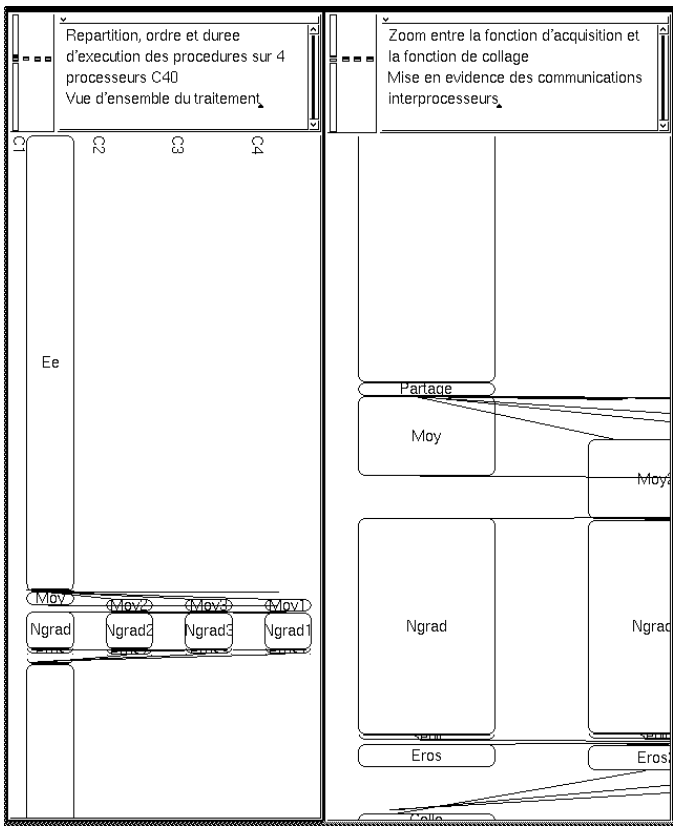


FIG. 11 – *Diagramme temporel: répartition des opérations sur les 4 C40 et zoom sur les communications interprocesseurs*

- un fichier C pour chaque processeur qui compose l'architecture (`test1.c`, `test2.c` s'il y a 2 processeurs TMS320C40 et que l'application s'appelle `test`). Chacun de ces fichiers, qui inclut `syndex.h` et `test.h`, doit être compilé et "linké" avec la librairie `syndex.lib` qui contient le noyau générique d'exécutif SynDEx,
- un fichier de configuration `test.cfg` qui décrit les processeurs et leurs inter-connections ainsi que les tâches qu'ils ont à exécuter. Ce fichier permet à un compilateur spécifique de regrouper les exécutables (compilés à partir des `test.c`) pour que le "loader" puisse charger et lancer ces exécutables sur les processeurs,
- un Makefile `test.C3L` qui automatise toutes les opérations de compilation pour l'environnement de compilation "3L Parallel C for C40" [15];

Il faut fournir au fichier make d'extension .C3L les fichiers qui contiennent les fonctions C que l'utilisateur a écrites, et qui sont les mêmes que celles appelées par le programme SIGNAL.

Etant donné que les entrées et sorties de notre application sont simulées dans notre cas par des fichiers séquentiels, les opérations d'entrées/sorties doivent être exécutées sur le processeur relié à l'hôte pour sous-traiter à ce dernier les accès fichiers (`fprintf` et `fscanf`). On a regroupé ces fonctions dans le fichier `inout.c`.

La commande de compilation de ce premier processeur est de ce fait particulière, donc la ligne de commande doit être explicitée dans le Makefile. Cette compilation nécessite :

- le programme C créé particulièrement pour ce processeur : `test.c`;
- le fichier contenant les fonctions C d'entrée/sortie : `fonction.c`;
- la librairie SynDEx contenant les fonctions de gestion des communications et des synchronisations. Cette librairie dépend du type de processeur utilisé, c'est la seule partie du logiciel SynDEx qui doit être modifiée quand on ajoute un nouveau processeur cible;
- le fichier de gestion des entêtes : `pgmhead.c`;
- le fichier des fonctions d'entrée/sortie : `inout.c`.

En ce qui concerne les autres processeurs, on peut sauter une ligne sans préciser de commande de compilation car elle est implicite. Les fichiers nécessaires à un processeur sont :

- le fichier contenant les fonctions C ;
- la librairie SynDEx ;
- son propre programme C généré par synDEx.

Dans le fichier make fourni par SynDEx, l'utilisateur est guidé (par les commentaires précédés de "#" et par les commandes de compilation pré-écrites) afin de composer lui-même le fichier make correspondant à son application.

Voici le fichier cont4q.C3L qui est le Makefile de la détection de contours sur 4 processeurs C40 :

```
# SynDEx v3.6b INRIA 1995/6/14-15:41:7
# application cont4q: no comment
# make file for 3L parallel C for C40

.c.obj: # rule to compile C source
        cl30 -qq -v40 -mb -mxx -x2 -p? $*
```

```
.obj.tsk: # rule to link processor task
        lnk30 -q -ar -c $** -lsyndex.lib -lc40csbs.lib -o $*.tsk

.cfg.app: # rule to produce multi-processor configuration
        config $*.cfg $*.app

# !!!!!!!!!!!!!!!!  WARNING  !!!!!!!!!!!!!!!!!!
# add here dependents to compile cont4q application library.
# For example, if your application source code
# is in the two files,
# file1.c and file2.c, add four lines
# (don't forget empty lines)
# and modify the cont4qLIB macro as follows :

# file1.obj: file1.c
#
# file2.obj: file2.c
#
# cont4qLIB = file1.obj file2.obj

# fichier de gestion des entetes
pgmhead.obj: pgmhead.c

# fichier des fonctions entrees/sorties
inout.obj: inout.c

# fichier des fonctions C appelees
fonction.obj: fonction.c

# chaque processeur a besoin de ce fichier
cont4qLIB = fonction.obj
# processors SC modules
cont4q1.obj: cont4q1.c cont4q.h

cont4q1.tsk: cont4q1.obj $(cont4qLIB)
             pgmhead.obj inout.obj
        lnk30 -q -ar -c $** -lsyndex.lib
                 -lc40cfbs.lib -o $*.tsk

# ligne de commande particuliere pour le processeur root
# fichier entrees/sorties car processeur d'interfacage
```

```
# fichier de gestion des entetes

cont4q2.obj: cont4q2.c cont4q.h
# programme C propre au processeur 2

cont4q2.tsk: cont4q2.obj $(cont4qLIB)

# programme C propre au processeur 3
cont4q3.obj: cont4q3.c cont4q.h

cont4q3.tsk: cont4q3.obj $(cont4qLIB)

# programme C propre au processeur 4
cont4q4.obj: cont4q4.c cont4q.h

cont4q4.tsk: cont4q4.obj $(cont4qLIB)

# multiprocessor network
cont4q.app: cont4q.cfg cont4q1.tsk cont4q2.tsk
            cont4q3.tsk cont4q4.tsk

cont4q.RUN:
        tis -:b $*.app

# That's all folks !!
```

7.7 Évaluation des performances temps réel

Une fois l'adéquation lancée avec les instructions de chronométrage [16], on génère les fichiers C relatifs à chaque processeur. Nous avons mesuré la durée d'exécution globale de la même application exécutée sur trois architectures différentes, sur un, deux, et quatre processeurs C40.

Comme les durées d'exécution des nœuds d'entrées/sorties correspondent à des accès fichiers pour notre application, elles n'ont pas été prises en compte car elles ne sont pas significatives. En effet, dans le cas du temps réel, les entrées/sorties de type caméra, écran sont beaucoup moins importantes et parfois n'interviennent pas quand les images sont prêtes dans l'un des deux tampons mémoire du "frame grabber" qui réalise l'acquisition.

Les durées d'exécution sont indiquées dans la deuxième colonne du tableau 2. La troisième colonne représente l'accélération effective quand on passe de 1 à n processeurs ($n = 1, 2, 4$). Enfin, la dernière colonne correspond à l'efficacité liée au changement d'architecture. C'est le rapport, exprimé en pourcentage, entre l'accélération effective et l'accélération optimale. Par exemple dans le cas de deux processeurs, le speed-up est de 1,89 ce qui correspond à

Implantation	Durée d'exécution	Speed-up	Efficacité
Sur un processeur	1,607 s	-	-
Sur deux processeurs	0,85 s	1,89	94,5 %
Sur quatre processeurs	0,478 s	3,35	83,75 %

TAB. 2 – *Mesures des durées d'exécution de l'application sur trois types d'architecture*

une efficacité de $\frac{1,89*100}{2} = 94,5\%$. La perte d'efficacité se situe au niveau des communications inter-processeurs. A la fin de ces mesures, l'algorithme peut être remis en cause si les contraintes temps réel que l'on s'était fixées ne sont pas satisfaites. Le passage d'une implantation à l'autre n'a pas nécessité de mise au point multi-processeur, une simple phase de chargement et d'édition de liens et de compilation a du être effectuée à chaque fois.

L'algorithme a été évalué sur deux types d'images de taille 100×160 : un "coyote" et une "vue aérienne de bâtiment". Les différents résultats obtenus après chaque étape du traitement (image initiale, moyenne, norme du gradient, seuillage et image finale) figurent dans l'annexe E.

8 Conclusion

Le but du travail présenté dans ce rapport consistait à montrer comment la méthodologie A^3 et le logiciel SynDEx pouvaient aider à l'implantation d'algorithmes de traitement d'images devant s'exécuter en temps réel. Afin d'illustrer les avantages de l'approche nous avons choisi une application classique en traitement d'images : la détection de contours. Ce qui nous importait ici n'était pas les performances de la détection de contours elle-même, mais plutôt les facilités apportées par le logiciel d'aide à l'implantation d'un tel algorithme représentatif du traitement d'images bas et moyen niveau.

Lors de l'implantation d'un tel algorithme on est confronté à plusieurs problèmes. Le premier concerne la puissance de calcul nécessaire pour l'exécuter en respectant les contraintes temps réel. Généralement une solution mono-processeur n'est pas suffisante. De plus une solution multi-processeur est parfois imposée lorsqu'on est contraint, par exemple, à délocaliser certains traitements près des capteurs et des actionneurs. Dans les deux cas on est conduit à résoudre le problème de la parallélisation de l'algorithme. Cela consiste premièrement à mettre en évidence les opérations potentiellement parallélisables, ceci de façon la plus indépendante possible des aspects matériels. Bien sûr par la suite il s'agit d'exploiter ce parallélisme potentiel en fonction des ressources matérielles dont on dispose tout en respectant les contraintes temps réel.

L'approche classique consistant à mettre au point un algorithme en mono-processeur puis à tenter de le porter sur une architecture multi-processeur est difficile à mettre en œuvre car elle demande, à la fois de paralléliser directement l'algorithme en fonction du multi-processeur (on perd de cette manière certains degrés de liberté mis en évidence par le parallélisme potentiel), et de s'assurer que les contraintes temps réel sont satisfaites en

exécutant l'algorithme en temps réel sur le multi-processeur cible. Cela conduit généralement à des temps de développements très longs car la mise au point est particulièrement complexe en multi-processeur. Cela est principalement dû à la diffficulté d'observer un état global lors d'une exécution distribuée pour laquelle il faut gérer efficacement les transferts de données (communications inter-processeur, synchronisations, partage de ressources ...). Si l'algorithme n'a pas été correctement parallélisé, c'est-à-dire si le découpage choisi a-priori n'est pas bien adapté à l'architecture cible, que celle-ci soit complètement imposée, ou soit partiellement imposée (seulement le type, mais pas le nombre de composants), si le système de communications inter-processeur n'est pas cohérent (inter-blocage), pour passer du constat d'un mauvais fonctionnement à une nouvelle version de l'implantation se comportant correctement, une phase de mise au point très longue est généralement nécessaire. Un autre problème se pose quand le comportement est correct en termes de valeurs et d'ordre des événements mais que les contraintes temps réel ne sont pas respectées. Il faut alors procéder à des modifications sur la parallélisation de l'algorithme, voire le modifier profondément, il faut aussi modifier l'architecture. On se retrouve alors dans la situation précédente.

A^3 et SynDEx, pour pallier à ces inconvénients, offrent trois caractéristiques importantes s'appuyant sur un formalisme unifié, celui des graphes (transformations) et des calculs (chemins, boucles critiques ...) qu'on peut leur appliquer. Tout d'abord, la spécification et la vérification d'algorithme à l'aide de graphes flot de données conditionnés respectant la sémantique des langages synchrones, d'une part laissent plus de liberté quant à l'expression du parallélisme (potentiel) et d'autre part permettent d'éviter les erreurs de mauvaises manipulations des variables ainsi que les erreurs de conception logiques, dues principalement à une mauvaise spécification du contrôle. Les modèles d'architectures caractérisées au bon niveau, que l'on utilise pour étiqueter le graphe de l'algorithme, permettent d'effectuer des choix de distribution et d'ordonnancement au maximum statiques conduisant à des surcoûts très faibles lors de l'exécution temps réel. Le choix de distributions et d'ordonnancements dynamiques n'est utilisé que lorsqu'il est incontournable; par exemple le choix des branches de calcul conditionnées est fait au moment ou le booléen de conditionnement est connu, c'est-à-dire lors de l'exécution. De plus, cela permet d'effectuer des prévisions de comportement afin de dimensionner au mieux dès le début les ressources matérielles. Enfin, ces choix faits à la compilation permettent de générer facilement les exécutifs qui gèreront un système de communications inter-processeur fiable et efficace (prévention des interblocages et calcul au plus juste des tampons mémoire). Le cycle d'implantation étant court, il est facile d'essayer plusieurs variantes de l'architecture matérielle en vue de respecter les contraintes temps réel. Enfin, cela permet de remettre en cause l'algorithme dans sa structure même, lorsqu'on ne peut pas respecter les contraintes.

Dans le cas de notre application de détection de contours, nous avons montré qu'il était facile et rapide de modifier à la fois l'algorithme et l'architecture pour obtenir les meilleures performances possibles avec les composants que nous avions à notre disposition, à savoir quatre TMS320C40. Le passage d'une implantation à l'autre, sur un, deux puis quatre processeurs s'effectue sans aucun effort. En particulier il n'y a pas eu de mise au point multi-processeur à effectuer. Cette approche permet d'assurer qu'un algorithme fonctionnant en

mono-processeur fonctionnera de la même manière en multi-processeur. Tout cela réduit de manière importante les temps de développements des application de traitement d'images s'exécutant en temps réel.

A Programme SIGNAL pipe-line

```
% TRAITEMENT GLOBAL D'UNE IMAGE EN LANGAGE SIGNAL :
   -moyenne
   -norme du gradient
   -seuillage
   -erosion binaire
%

% programme vicard/signal/source.sig  %

process contour = (integer m,n,p;
  [p,p] integer M5,M1,M2,M3,M4 % masques des convolutions %
              )
  { ? [m,n] integer E; integer seuil2
    ! [m,n] integer SE, S1,S2,S3,S4,S5  }

  (| cons{seuil2}
   | SE:= E
   | S1:= moyenne(m,n,p,M5){E}
   | S2:= gradient(m,n,p,M1,M2,M3,M4){S1}
   | S3:= seuillage(m,n){seuil2, S2}
   | S4:= erosion(m,n,p){S3}
   | S5:= erosion(m,n,p){S4}
   |)
where

  process bordure = (integer m,n)
  { ? [m,n]integer E
    ! [m+2,n+2]integer S }
  (| S := [{x to m, y to n}:[x+1,y+1]:E[x,y],
           {x to m}:[x+1,1]:E[x,1],   [1,1]:E[1,1],
           {x to m}:[x+1,n+1]:E[x,n], [m+1,n+1]:E[m,n],
           {y to n}:[1,y+1]:E[1,y],   [1,n+1]:E[1,n],
           {y to n}:[m+1,y+1]:E[m,y], [m+1,1]:E[m,1]
          ]
  |)
  end
;
  process convolution = (integer m,n,p; [p,p]integer Mask)
  { ? integer x,y; [m+2,n+2]integer E
    ! integer z }
```

```
  (| array i to p of
     (| array j to p of
        (| res1:= res1 + Mask[i,j] * E[(x+i)-1,(y+j)-1]
         |)
        with res1[0]:zero1
        end
      | zero1:= 0
      | res2:= res2+res1[p]
      |)
     with res2[0]:zero2
     end
   | zero2:= 0
   | z:= res2[p]
   |)
  where [p,p]integer res1; [p]integer zero1, res2;
            integer zero2
  end
;
  process moyenne = (integer m,n,p; [p,p]integer M5)
  { ? [m,n]integer E
    ! [m,n]integer S }
% S:= moyenne{E}
%  where function moyenne={? [m,n]integer E ! [m,n]integer S}
% (| Z := bordure(m,n){E}
   | array x to m of  array y to n of
     (| v:= convolution (m,n,p,M5) {x,y,Z}
      | S:= v/9
      |)
     end  end
   |)
  where [m,n]integer v; [m+2,n+2]integer Z
  end
;
  process gradient = (integer m,n,p;
                      [p,p]integer M1, M2, M3, M4)
  { ? [m,n]integer E
    ! [m,n]integer S }
% S:= gradient{E}
%  where function gradient={? [m,n]integer E ! [m,n]integer S}
% (| Z := bordure(m,n){E}
   | array x to m of  array y to n of
     (| g1:= convolution (m,n,p,M1){x,y,Z}
```

```
      | g2:= convolution (m,n,p,M2){x,y,Z}
      | g3:= convolution (m,n,p,M3){x,y,Z}
      | g4:= convolution (m,n,p,M4){x,y,Z}
      | S := g1*g1 + g2*g2 + g3*g3 + g4*g4
      |)
    end  end
  |)
  where [m+2,n+2]integer Z; [m,n]integer g1,g2,g3,g4
  end
;
  process seuillage = (integer m,n)
  { ? integer seuil2; [m,n]integer E
    ! [m,n]integer S }
  (| array i to m of  array j to n of
     (| o:= E[i,j] > seuil2
      | S:= (1 when o) default (0 when not o)
      |)
    end  end
  |)
  where [m,n]logical o
  end
;
  process erosion = (integer m,n,p)
  { ? [m,n]integer E
    ! [m,n]integer S }
  (| Z:= bordure(m,n){E}
   | array x to m of  array y to n of
     (| array i to (p-2) of
        (| array j to (p-1) of
           res1:= res1 and Z[(x+i)-1,(y+j)-1]/=0
           with res1[0]:zero1 end
         | zero1:= true
         | res2:= res2 and res1[p-1]
         |)
        with res2[0]:zero2 end
      | zero2:= true
      | S:= 1 when res2[p-2] default 0 when not res2[p-2]
      |)
    end  end
  |)
  where [m+2,n+2]integer Z;
    [m,n,(p-2),(p-1)]logical res1;
```

```
% [m,n,p    ] logical zero1;
% [m,n,(p-2)] logical res2,zero1; [m,n]logical zero2
  end

end
```

B Programme SIGNAL et C, pipe-line et data-parallel

B.1 Programme SIGNAL

```
% programme vicard/quadrip/quadrip.sig   %

process contour4c = (integer m,n,p;
                     [p,p] integer M5,M1,M2,M3,M4
                    )
   { ? [m,n] integer E;integer seuil2
     ! [m,n] integer GS
   }

   (|cons{seuil2}
    |{E1,u1h,u1b} := creation{E,0}
    |{E2,u2h,u2b} := creation{E,m/4}
    |{E3,u3h,u3b} := creation{E,m/2}
    |{E4,u4h,u4b} := creation{E,(3*m)/4}

    |{S1,u1sh,u1sb} := pmoy{E1,u1h,u2h}
    |{S2,u2sh,u2sb} := pmoy{E2,u1b,u3h}
    |{S3,u3sh,u3sb} := pmoy{E3,u2b,u4h}
    |{S4,u4sh,u4sb} := pmoy{E4,u3b,u4b}

    |{E11} := pgradient{S1,u1sh,u2sh}
    |{E22} := pgradient{S2,u1sb,u3sh}
    |{E33} := pgradient{S3,u2sb,u4sh}
    |{E44} := pgradient{S4,u3sb,u4sb}

    |{S11,u11h,u11b} := pseuil{seuil2,E11}
    |{S22,u22h,u22b} := pseuil{seuil2,E22}
    |{S33,u33h,u33b} := pseuil{seuil2,E33}
    |{S44,u44h,u44b} := pseuil{seuil2,E44}

    |{S111} := perosion{S11,u11h,u22h}
```

```
    |{S222} := perosion{S22,u11b,u33h}
    |{S333} := perosion{S33,u22b,u44h}
    |{S444} := perosion{S44,u33b,u44b}
    |{GS} := collage{S111,S222,S333,S444}
    |)

 where [n] integer u1h,u1b,u2h,u2b,u3h,u3b,u4h,u4b ;
       [n] integer u11h,u11b,u22h,u22b,
                   u33h,u33b,u44h,u44b ;
       [n] integer u1sh,u1sb,u2sh,u2sb,
                   u3sh,u3sb,u4sh,u4sb;
       [m/4,n] integer E1,E11,E2,E22,E3,E33,
                       E4,E44,S1,S11,S111,
                       S2,S22,S222,S3,S33,
                       S333,S4,S44,S444
       function creation =  {? [m,n  ] integer GE;
                                       integer trans
                            ! [m/4,n] integer PS;
                              [n    ] integer uh,ub};
       function pmoy =      {? [m/4,n] integer PE;
                              [n    ] integer uh,ub
                            ! [m/4,n] integer PS;
                              [n    ] integer ush,usb};
       function pgradient = {? [m/4,n] integer PE;
                              [n    ] integer uh,ub
                            ! [m/4,n] integer PS};
       function pseuil =    {?  integer seuil;
                              [m/4,n] integer PE
                            ! [m/4,n] integer PS;
                              [n    ] integer uh,ub};
       function perosion =  {? [m/4,n] integer PE;
                              [n    ] integer uh,ub
                            ! [m/4,n] integer PS};
       function collage =   {? [m/4,n] integer PE1,
                              PE2,PE3,PE4
                            ! [m,n  ] integer GS}

end
```

B.2 Procédures C appelées par le programme SIGNAL

```
/* #define BI 1   pour SIGNAL   cc -DBI=1 */
/* #define BI 0   pour SynDEx   cc -DBI=0 */


#include <stdio.h>

#define m 100   /* Nombre de lignes de la matrice */
                /* representant l'image */
#define n 160   /* Nombre de colonnes   */
#define p 3     /* Taille des masques   */

/*********************************************/
/*  Contenu des differents masques           */
/*********************************************/
   /* Moyenne             */
   int M5[p][p] = {{1,1,1},{1,1,1},{1,1,1}};
   /* Gradient vertical   */
   int M1[p][p] = {{1,0,-1},{1,0,-1},{1,0,-1}};
   /* Gradient horizontal */
      int M2[p][p] = {{-1,-1,-1},{0,0,0},{1,1,1}};
   /* Gradient oblique    */
   int M3[p][p] = {{-1,-1,0},{-1,0,1},{0,1,1}};
   /* Gradient oblique    */
   int M4[p][p] = {{0,-1,-1},{1,0,-1},{1,1,0}};
/*************************************************
*  Creation d'une imagette
*  Entree : E(m,n) entier  = image de depart
*           trans  entier  = valeur de la translation
*                  - trans =    0 imagette superieur
*                  - trans =  m/4 imagette directement
*                             en dessous
*                  - trans =  m/2 avant derniere imagette
*                  - trans = 3m/4 imagette inferieure
*  Sortie : S(m/4,n) entier imagette
*           uh(n)    entier = premiere ligne de S
*           ub(n)    entier = derniere ligne de S
*************************************************/
 creation(E,trans,S,uh,ub)
  int E[BI+m][BI+n], trans;
  int S[BI+m/4][BI+n], uh[BI+n], ub[BI+n];
  { int i,j;
    for (i=0; i<m/4; i++) for (j=0; j<n; j++)
```

```
      S[BI+i][BI+j] = E[BI+i+trans][BI+j];
    for (j=0; j<n; j++)
     { uh[BI+j] = S[BI+0][BI+j];
       ub[BI+j] = S[BI+m/4-1][BI+j];
     }
  }
/*****************************************************
*  Creation des bords d'une imagette
*  Entree : E(m/4,n) entier =  imagette de depart
*           uh(n)    entier =  premiere ligne de l'imagette
*                    directement en dessous
*                    (pour la derniere imagette
*                    ca correspond a sa derniere ligne)
*           ub(n)    entier = derniere ligne de l'imagette
*                    directement au dessus
*                    (pour la premiere imagette
*                    ca correspond a sa premiere ligne)
*  Sortie : B(m/4+2,n+2) entier = imagette + bords
*****************************************************/
 bord(E,uh,ub,B)
  int E[BI+m/4][BI+n], uh[BI+n], ub[BI+n] ;
  int B[BI+m/4+2][BI+n+2];
  { int i,j;
    /*Remplissage interieur */
    for (i=0; i<m/4; i++) for(j=0; j<n; j++)
        B[BI+i+1][BI+j+1] = E[BI+i][BI+j];
    /*Remplissage miroir des bords verticaux */
    for (i=0; i<m/4; i++)
      { B[BI+i+1][BI+0  ] = E[BI+i][BI+0];
        B[BI+i+1][BI+n+1] = E[BI+i][BI+n-1];
      }
   /*Remplissage miroir des bords horizontaux */
    for (j=0; j<n; j++)
      { B[BI+0    ][BI+j+1] = uh[BI+j];
        B[BI+m/4+1][BI+j+1] = ub[BI+j];
      }
   /*Remplissage des coins */
    B[BI+0][BI+0      ] = E[BI+0    ][BI+0  ];
    B[BI+0][BI+n+1    ] = E[BI+0    ][BI+n-1];
    B[BI+m/4+1][BI+0  ] = E[BI+m/4-1][BI+0  ];
    B[BI+m/4+1][BI+n+1] = E[BI+m/4-1][BI+n-1];
  }
```

```
/*****************************************************
*  Moyenne de chaque imagette
*  Entree : E[BI+m/4][BI+n] entier imagette etudiee
*           uh(n)    entier =  premiere ligne de l'imagette
*                    directement en dessous
*                    (pour la derniere imagette
*                    ca correspond a sa derniere ligne
*           ub(n)    entier = derniere ligne de l'imagette
*                    directement au dessus
*                    (pour la premiere imagette
*                    ca correspond a sa premiere ligne
*  Sortie : S[BI+m/4][BI+n] entier =  imagette moyennee
*           uh(n) entier = premiere ligne de S
*           ub(n) entier = derniere ligne de S
*****************************************************/
 void pmoy(E,uh,ub,S)
  int E[BI+m/4][BI+n],uh[BI+n], ub[BI+n], S[BI+m/4][BI+n];
  { int B[BI+m/4+2][BI+n+2];
    int i,j;
    bord(E,uh,ub,B);
    for (i=0; i<m/4; i++) for (j=0; j<n; j++)
      S[BI+i][BI+j]= (convolution (i,j,B,M5)/9);
    for (j=0; j<n; j++)


    { uh[BI+j] = S[BI+0][BI+j];
      ub[BI+j] = S[BI+m/4-1][BI+j];
    }
  }

 int  convolution(x,y,E,Mask)
  int x,y,E[BI+m/4+2][BI+n+2],Mask[p][p];
  { int i,j,z;
    for(z=0, i=0; i<p; i++)  for(j=0; j<p; j++)
      z = z+(Mask[i][j] * E[BI+(x+i)][BI+(y+j)]);
    return z;
  }
/*****************************************************
*  Norme du gradient dans les 4 directions
*  pour une imagette
*  Entree : E[BI+m/4][BI+n] entier imagette etudiee
```

```
*           uh(n)    entier =  premiere ligne de l'imagette
*                    directement en dessous
*                    (pour la derniere imagette
*                    ca correspond a sa derniere ligne
*           ub(n)    entier = derniere ligne de l'imagette
*                    directement au dessus
*                    (pour la premiere imagette
*                    ca correspond a sa premiere ligne
*  Sortie : S[BI+m/4][BI+n] entier norme du gradient
*                           de l'imagette
*****************************************************/
 pgradient(E,uh,ub,S)
  int E[BI+m/4][BI+n],uh[BI+n], ub[BI+n];
  int S[BI+m/4][BI+n];
  { int B[BI+m/4+2][BI+n+2];
    int x,y,j;
    int G1, G2, G3, G4;
    bord(E,uh,ub,B);
    for (x=0; x<m/4; x++) for (y=0; y<n; y++)
    { /* convolution ds 4 directions */
      G1 = convolution (x,y,B,M1);
      G2 = convolution (x,y,B,M2);
      G3 = convolution (x,y,B,M3);
      G4 = convolution (x,y,B,M4);
      S [BI+x][BI+y] = G1*G1+G2*G2+G3*G3+G4*G4;
     }/* norme euclidienne au carre */
  }
/*******************************************************
*  Seuillage de chaque imagette
*  Entree : seuil2 entier = valeur du seuil
*           E[BI+m/4][BI+n] entier imagette
*  Sortie : S[BI+m/4][BI+n] entier imagette seuillee
*           uh(n) entier = premiere ligne de S
*           ub(n) entier = derniere ligne de S
*******************************************************/
 pseuil(seuil2,E,uh,ub,S)
  int seuil2, E[BI+m/4][BI+n];
  int uh[BI+n], ub[BI+n], S[BI+m/4][BI+n];
  { int i,j;
    for(i=0; i<m/4; i++) for(j=0; j<n; j++)
      S[BI+i][BI+j] = (E[BI+i][BI+j] > seuil2);
    for (j=0; j<n; j++)
```

```
    { uh[BI+j] = S[BI+0][BI+j];
      ub[BI+j] = S[BI+m/4-1][BI+j];
    }
  }
/*******************************************************
*  Erosion de chaque imagette +bords
*  Entree : E[BI+m/4][BI+n] entier imagette etudiee
*           uh(n)    entier =  premiere ligne de l'imagette
*                    directement en dessous
*                    (pour la derniere imagette
*                    ca correspond a sa derniere ligne
*           ub(n)    entier = derniere ligne de l'imagette
*                    directement au dessus
*                    (pour la premiere imagette
*                    ca correspond a sa premiere ligne
*  Sortie : S[BI+m/4][BI+n] entier imagette erodee
*******************************************************/
 perosion(E,uh,ub,S)
  int E[BI+m/4][BI+n], uh[BI+n], ub[BI+n];
  int S[BI+m/4][BI+n];
  { int B[BI+m/4+2][BI+n+2];
    int x,y,i,j;
    int res;
    bord(E,uh,ub,B);
    for (x=0; x<m/4; x++) for(y=0; y<n; y++)
    { for (res=1, i=0; i<p-2; i++) for (j=0; j<p-1; j++)
        res = res && (B[BI+x+i][BI+y+j] !=0);
        S[BI+x][BI+y]=res;
        /* est-ce que !=0 est utile ? */
    }
  }
/*****************************************
*  Recollage des 4 imagettes
*  Entree : E1(m/4,n),E2(m/4,n),E3(m/4,n),
*           E4(m/4,n) entier
*  Sortie : S(m,n) entier
*****************************************/
 collage (E1,E2,E3,E4,S)
  int E1[BI+m/4][BI+n], E2[BI+m/4][BI+n],
      E3[BI+m/4][BI+n], E4[BI+m/4][BI+n];
  int S[BI+m][BI+n];
  { int i,j;
```

```
    for (i=0; i<m/4; i++) for(j=0; j<n; j++)
     { S[BI+i]      [BI+j] = E1[BI+i][BI+j];
       S[BI+i+m/4]  [BI+j] = E2[BI+i][BI+j];
       S[BI+i+m/2]  [BI+j] = E3[BI+i][BI+j];
       S[BI+i+3*m/4][BI+j] = E4[BI+i][BI+j];
      }
   }
/*****************************************/
```

B.3 Programmes C générés par le compilateur SIGNAL

B.3.1 Le fichier _E.c

```
#include <stdio.h>

typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0

int i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;

FILE
    *fre_11,
    *frseuil2_12,
    *fwgs_13;

extern void begio()
{
 fre_11 = fopen("RE.dat","r");
 if (!fre_11)
 {fprintf(stderr,"Cannot open file %s\n ","RE.dat");
 exit(1);}
 rheader(fre_11,160,100,"RE.dat");
 frseuil2_12 = fopen("RSEUIL2.dat","r");
 if (!frseuil2_12)
 {fprintf(stderr,"Cannot open file %s\n ","RSEUIL2.dat");
 exit(1);}
 fwgs_13 = fopen("WGS.dat","w");
 if (!fwgs_13)
 {fprintf(stderr,"Cannot open file %s\n ","WGS.dat");
 exit(1);}
```

```
 wheader(fwgs_13,160,100);
}

extern void endio()
{
 fclose(fre_11);
 fclose(frseuil2_12);
 fclose(fwgs_13);
}

extern void re(e_11,h_4_h)
int e_11[101][161];
event *h_4_h;
{
 for (i1 = 1;i1<=100;i1++)
  for (i2 = 1;i2<=160;i2++)
*h_4_h = (fscanf(fre_11,"%d",&e_11[i1][i2])!=EOF);
}
extern void rseuil2(seuil2_12,h_4_h)
int *seuil2_12;
event *h_4_h;
{
 *h_4_h = (fscanf(frseuil2_12,"%d",seuil2_12)!=EOF);
}

extern void wgs(gs_13)
int gs_13[101][161];
{
 for (i1 = 1;i1<=100;i1++)
  for (i2 = 1;i2<=160;i2++)
fprintf(fwgs_13,"%d\n",gs_13[i1][i2]*255);
}
```

B.3.2 Le fichier _M.c

```
typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0
```

```
extern logical ccontour4c();
extern logical icontour4c();
extern void begio();
extern void endio();

extern int main()
{
 logical code;
 begio();
 code = icontour4c();
 while(code)code = ccontour4c();
 endio();return 0;
}
```

B.3.3 Le fichier _S.c

```
typedef int event;
typedef int logical;

#define TRUE 1
#define FALSE 0

int i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;

extern void collage();
extern void perosion();
extern void pseuil();
extern void pgradient();
extern void pmoy();
extern void creation();

extern void re();
extern void rseuil2();

extern void wgs();

/*C  Declaration of signals */

int m4_10[4][4], m3_9[4][4], m2_8[4][4], m1_7[4][4],
    m5_6 [4][4],
    xzx_1, xzx_2
```

```
int e_11[101][161], seuil2_12;
int gs_13[101][161];
int u1h_58[161],   u1b_59[161],
    u2h_60[161],   u2b_61[161],
    u3h_62[161],   u3b_63[161],
    u4h_64[161],   u4b_65[161],
    u11h_66[161], u11b_67[161],
    u22h_68[161], u22b_69[161],
    u33h_70[161], u33b_71[161],
    u44h_72[161], u44b_73[161],
    u1sh_74[161], u1sb_75[161],
    u2sh_76[161], u2sb_77[161],
    u3sh_78[161], u3sb_79[161],
    u4sh_80[161], u4sb_81[161],
    e1_82[26][161], e11_83[26][161],
    e2_84[26][161], e22_85[26][161],
    e3_86[26][161], e33_87[26][161],
    e4_88[2 161], e44_89[26][161],
    s1_90[26][161], s11_91[26][161],
    s111_92[26][161],
    s2_93[26][161], s22_94[26][161],
    s222_95[26][161],
    s3_96[26][161], s33_97 ][161],
    s333_98[26][161],
    s4_99[26][161], s44_100[26][161],
    s444_101[26][ ],
    trans_103, trans_104, trans_105, trans_106,
    seuil_107, seuil_108,
    seuil_109, seuil_110;
/*C  Declaration of clocks */

event h_4_h, h_11_h;

/*C  Body of the initialization procedure */

extern logical icontour4c()
{int xzx_1,xzx_2;
 m4_10[1][1] = 0;
 m4_10[1][2] = -1;
 m4_10[1][3] = -1;
 m4_10[2][1] = 1;
 m4_10[2][2] = 0;
```

```
 m4_10[2][3] = -1;
 m4_10[3][1] = 1;
 m4_10[3][2] = 1;
 m4_10[3][3] = 0;
 m3_9[1][1] = -1;
 m3_9[1][2] = -1;
 m3_9[1][3] = 0;
 m3_9[2][1] = -1;
 m3_9[2][2] = 0;
 m3_9[2][3] = 1;
 m3_9[3][1] = 0;
 m3_9[3][2] = 1;
 m3_9[3][3] = 1;
 m2_8[1][1] = -1;
 m2_8[1][2] = -1;
 m2_8[1][3] = -1;
 m2_8[2][1] = 0;
 m2_8[2][2] = 0;
 m2_8[2][3] = 0;
 m2_8[3][1] = 1;
 m2_8[3][2] = 1;
 m2_8[3][3] = 1;
 m1_7[1][1] = 1;
 m1_7[1][2] = 0;
 m1_7[1][3] = -1;
 m1_7[2][1] = 1;
 m1_7[2][2] = 0;
 m1_7[2][3] = -1;
 m1_7[3][1] = 1;
 m1_7[3][2] = 0;
 m1_7[3][3] = -1;
 for (xzx_1 = 1;xzx_1<=3;xzx_1++)
  for (xzx_2 = 1;xzx_2<=3;xzx_2++)
m5_6[xzx_1][xzx_2] = 1;
 rseuil2(&seuil2_12,&h_4_h);
 if (!h_4_h) return FALSE;

 trans_103 = 0;
 trans_104 = 100 / 4;
 trans_105 = 100 / 2;
 trans_106 = (3 * 100) / 4;
 seuil_107 = seuil2_12;
```

```
 seuil_108 = seuil2_12;
 seuil_109 = seuil2_12;
 seuil_110 = seuil2_12;
 return TRUE;
}

/*C  Body of the program */

extern logical ccontour4c()
{
 h_11_h = TRUE;
 re(e_11,&h_4_h);
 if (!h_4_h) return FALSE;

 creation(e_11,0,e1_82,u1h_58,u1b_59);
 creation(e_11,25,e2_84,u2h_60,u2b_61);
 creation(e_11,50,e3_86,u3h_62,u3b_63);
 creation(e_11,75,e4_88,u4h_64,u4b_65);
 pmoy(e1_82,u1h_58,u2h_60,s1_90,u1sh_74,u1sb_75);
 pmoy(e2_84,u1b_59,u3h_62,s2_93,u2sh_76,u2sb_77);
 pmoy(e3_86,u2b_61,u4h_64,s3_96,u3sh_78,u3sb_79);
 pmoy(e4_88,u3b_63,u4b_65,s4_99,u4sh_80,u4sb_81);
 pgradient(s1_90,u1sh_74,u2sh_76,e11_83);
 pgradient(s2_93,u1sb_75,u3sh_78,e22_85);
 pgradient(s3_96,u2sb_77,u4sh_80,e33_87);
 pgradient(s4_99,u3sb_79,u4sb_81,e44_89);
 pseuil(seuil2_12,e11_83,s11_91,u11h_66,u11b_67);
 pseuil(seuil2_12,e22_85,s22_94,u22h_68,u22b_69);
 pseuil(seuil2_12,e33_87,s33_97,u33h_70,u33b_71);
 pseuil(seuil2_12,e44_89,s44_100,u44h_72,u44b_73);
 perosion(s11_91,u11h_66,u22h_68,s111_92);
 perosion(s22_94,u11b_67,u33h_70,s222_95);
 perosion(s33_97,u22b_69,u44h_72,s333_98);
 perosion(s44_100,u33b_71,u44b_73,s444_101);
 collage(s111_92,s222_95,s333_98,s444_101,gs_13);
 wgs(gs_13);


 return TRUE;
}
```

C Programme de gestion des entêtes

```
/* lecture et ecriture de l'entete d'un fichier .PGM */

/* pgmheader.c */
#include<stdio.h>

/* nextn saute les commentaires
(commencant par # et termine par LF)
et convertit un ascii-decimal en entier positif
ou retourne -1 si erreur
*/

int nextn(f)
FILE *f;
{char c;
 int n=0;
 while(!isdigit(c=getc(f)))
 { if(c=='#') while(getc(f)!=0x0a);
else if(!isspace(c)) return -1;
 }
 while(isdigit(c))
 { n=10*n+(c-'0');
c=getc(f);
 }
 if(!isspace(c)) n=-1;
 return n;
}

/* rheader teste si l'entete est correcte  */
extern void rheader(f,width,height,filename)
FILE *f;
int width,height;
char *filename;
{int err=0;
 int n;
 err |= getc(f)!='P';
 err |= getc(f)!='2';
 err |= !isspace(getc(f));
 err |= nextn(f)!=width;
 err |= nextn(f)!=height;
 err |= nextn(f)!=255;
 if(err)
```

```
 {fprintf(stderr,"Entete incorrecte %s\n",filename);
 exit(1);}
}


/* wheader cree une nouvelle entete */
extern void wheader(f,width,height)
FILE *f;
int width,height;
{fprintf(f,"P2\n%d %d\n255\n", width, height);
}

/*
main()
{int err, w, h, line, col;
 err |= getc(stdin)!='P';
 err |= getc(stdin)!='5';
 err |= !isspace(getc(stdin));
 err |= (w=nextn(stdin))<0;
 err |= (h=nextn(stdin))<0;
 err |= nextn(stdin)!=255;
 if(err)
 {fprintf(stderr,"Entete incorrecte\n");
 exit(1);}
 wheader(stdout,w,h);
 for(line=0; line<h; line++)
 { for(col=0; col<w; col++)
printf(" %d", getc(stdin));

printf("\n");
 }
 return 0;
}
*/
```

D Graphes SynDEx pour l'application détection de contours

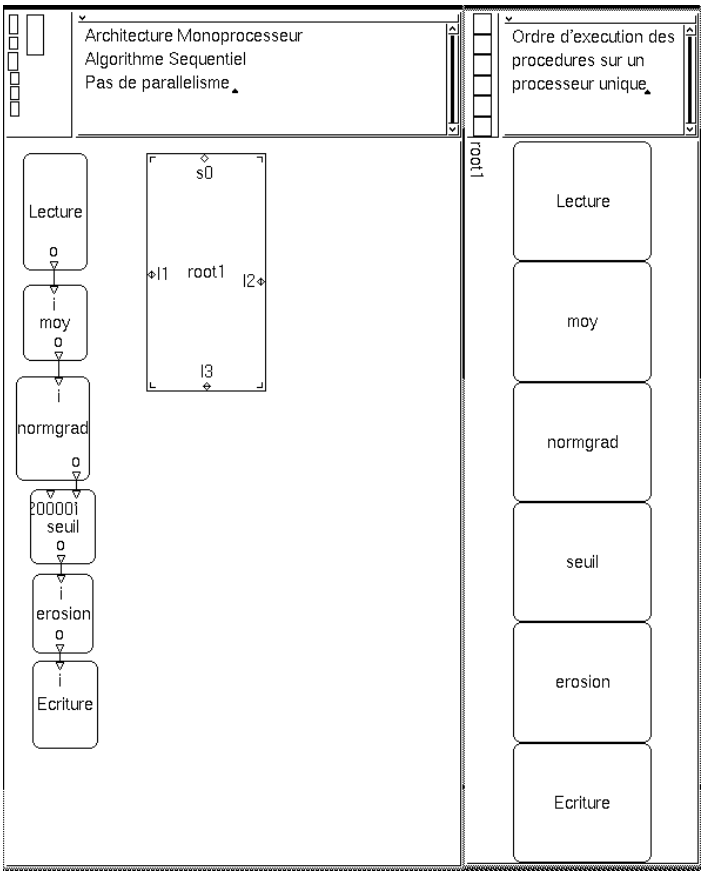


FIG. 12 – *Vue topologique et temporelle de l'implantation monoprocesseur de la version séquentielle de l'algorithme*

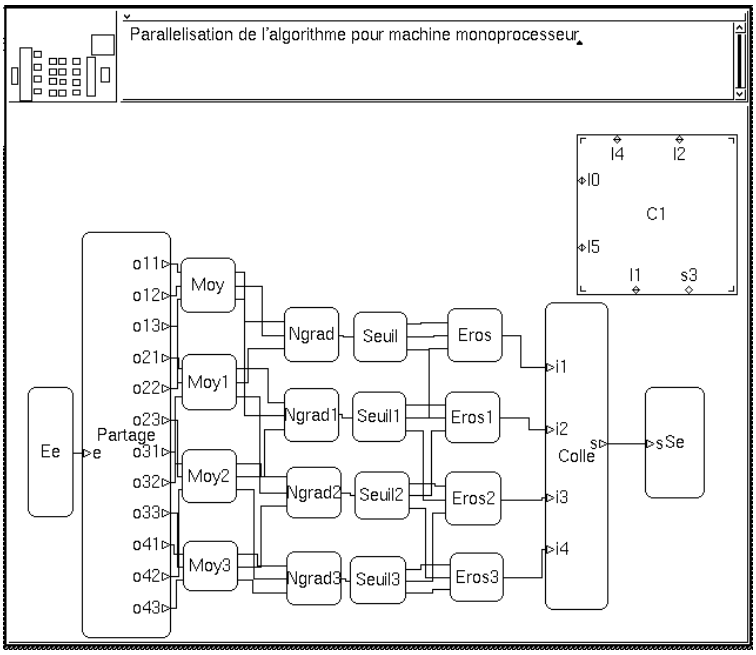


FIG. 13 – *Vue topologique de l'implantation monoprocesseur de la version parallélisée de l'algorithme*

E Visualisation des résultats

Exemples d'images de taille 160 x 100 au format PGM

FIG. 14 – *Image initiale*

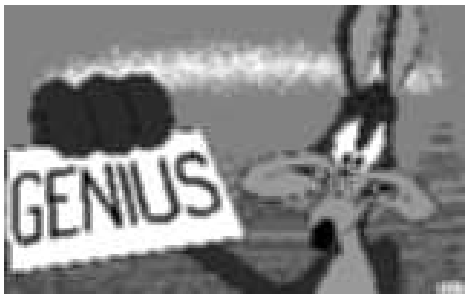


FIG. 15 – *Moyenne*

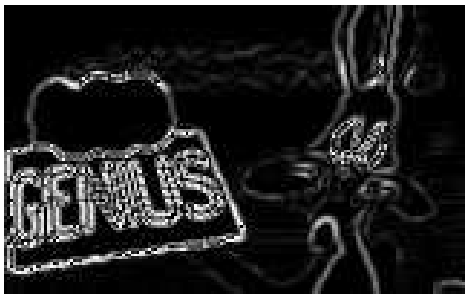


FIG. 16 – *Norme du gradient*

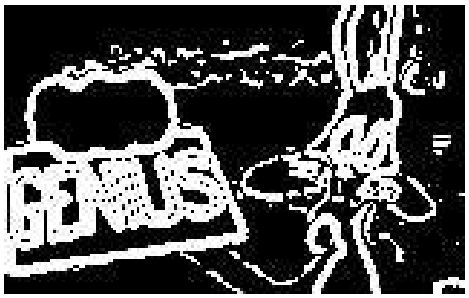


FIG. 17 – *Seuillage*

FIG. 18 – *Erosion*

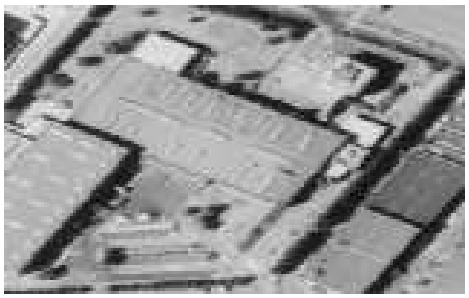


FIG. 19 – *Image initiale*

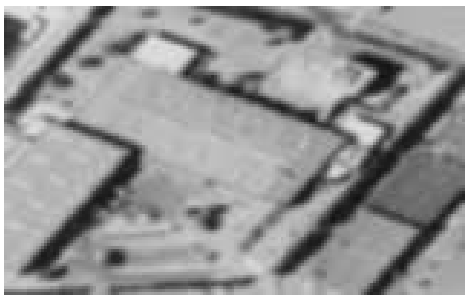


FIG. 20 – *Moyenne*

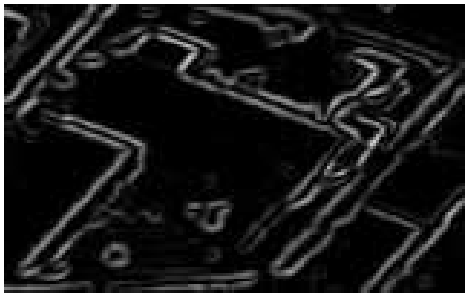


FIG. 21 – *Norme du gradient*

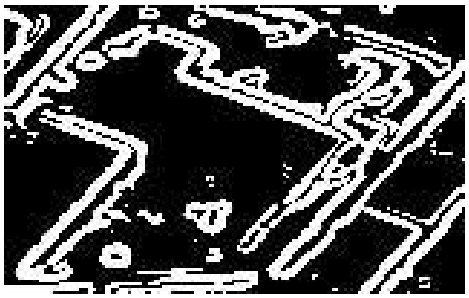


FIG. 22 – *Seuillage*

FIG. 23 – *Erosion*

Références

[1] Y. Sorel
Massively Parallel Computing Systems with Real Time Constraints
The " Algorithm Architecture Adequation " Methodology
Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing-Ischia Italy
May 1994.

[2] C. Lavarenne, Y. Sorel
Performance Optimization of Multiprocessor Real-Time Applications by Graph Transformations.
Parallel Computing 93, Grenoble, Septembre 1993.

[3] M.Gondran, M.Minoux.
Graphes et algorithmes.
Eyrolles 1969.

[4] J. Zwiers and W. Janssen
Partial order based design of concurrency systems
In a Decade of Concurrency, reflexions and perspectives, REX School/Symposium,pages 622-684,Springer Verlag, June 1993.

[5] B. Charron-Bost
Mesures de la concurrence et du parallélisme des calculs répartis.
Thèse de Doctorat en Informatique
Université Paris VII. Septembre 1989.

[6] *TMS320C4x User's Guide*
Texas Instruments 1993.

[7] J.P. Cocquerez, S. Philipp, P. Bolon, J.M. Chassery
Analyse d'images: filtrage et segmentation
Masson, Collection Enseignement de la physique, 1995.

[8] A. Benveniste, G. Berry
the synchronous approach to reactive and real-time systems.
Proceedings of the IEEE, 79(9):1270-1282, Sep. (1991)

[9] P. Leguernic, T. Gautier, M. Leborgne, C. Lemaire
Programming real-time applications with SIGNAL.
Proceedings of the IEEE, 79(9):1321-1336, Sep. (1991)

[10] C. Lavarenne, R. Reynaud, Y. Sorel
Spécification et validation à l'aide d'un langage synchrone d'un protocole d'appariement de données asynchrones.
Quartozième Colloque GRETSI, Juan-Les-Pins, Septembre 1993.

[11] C. Lavarenne, Y. Sorel
Documentation en ligne imprimable du logiciel SynDEx d'aide à l'implantation d'algorithmes sur architectures multi-processeurs sous contraintes temps-réel

[12] C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine:
The SynDEx software environment for real-time distributed systems design and implementation.
Proc. of the European Control Conference, 1991.

[13] P. Bournai, C. Lavarenne, P. Le Guernic, O. Maffeïs, Y. Sorel
Interface SIGNAL-SynDEx.
Rapports de Recherche INRIA n^o 2206, 1994.

[14] C. Lavarenne, Y. Sorel:
Specification, Performance Optimization and Executive Generation for Real-Time Embedded Multiprocessor Applications with SynDEx.
Proc. of Real-Time Embedded Processing for Space Applications, CNES International Symposium, 1992.

[15] *Parallel C User Guide for Texas Instruments TMS320C40 Version 1.0.2*
Société 3L Ltd, 1992.

[16] F. Ennesser, C. Lavarenne, Y. Sorel:
Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.
INRIA Research Report n°1769, 1992.

Table des matières

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LES NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399