UTILISATION DE SQUELETTES FONCTIONNELS AU SEIN D'UN OUTIL D'AIDE A LA PARALLELISATION

Dominique GINHAC, Jocelyn SEROT, Jean Pierre DERUTIN

LASMEA - UMR 6602 CNRS, Campus des Cézeaux, 63177 Aubiere
e-mail: (ginhac,jserot,derutin)@lasmea.univ-bpclermont.fr

RESUME - L'objet de ce papier est de présenter les premiers travaux relatifs à la création d'un outil automatique d'aide à la parallélisation en vue d'effectuer du prototypage rapide d'applications de vision artificielle sur machine MIMD à mémoire distribuée. L'approche proposée repose sur l'encapsulation de schémas de parallélisation usuels sous la forme de squelettes admettant deux définitions équivalentes. La première, interprétable par un langage fonctionnel, permet la vérification formelle des programmes, la seconde décrit l'implantation sous la forme de graphe pseudo-flot de données paramétrable.

ABSTRACT - In this paper, we present the first steps towards a software environment dedicated to the automatic parallelisation of real-time vision algorithms for a MIMD-DM machine. The approach is based upon the notion of algorithmic skeletons, the goal of which is to encapsulate all the aspects of a given parallelization scheme. Each skeleton has two definitions : the first one allows parallel programs to be prototyped on sequential architecture, the second one describes its implementation as a pseudo-data flow graph on the actual parallel hardware.

1 Introduction

Malgré la puissance sans cesse accrue des machines séquentielles, les systèmes complexes de vision artificielle nécessitent souvent l'utilisation d'architectures parallèles dédiées afin de répondre aux exigences temporelles.

Toutefois, la programmation efficace des architectures multi-processeurs de type MIMD à mémoire distribuée (par exemple la machine Transvision [7] à base de Transputers T9000) se heurte à deux difficultés. Premièrement, au niveau spécification, on doit expliciter le parallélisme potentiel des applications en les décomposant en tâches concurrentes et communiquantes. Deuxièmement, l'application doit être placée sur le réseau physique de processeurs. Cette phase consiste à réduire le parallélisme potentiel explicité par le programmeur à celui disponible sur l'architecture cible. Cela implique de distribuer et d'ordonnancer les programmes sur les processeurs tout en assurant une charge de travail homogène sur l'ensemble des processeurs. Cela entraine également une gestion purement explicite des communications (avec éventuellement un contrôle du routage des données si les processeurs ne sont pas directement connectés).

Une gestion purement manuelle de l'ensemble de ces tâches augmente fortement la complexité de programmation des architectures parallèles, entrainant de ce fait des temps de

cycle conception-implantation-validation importants. De fait, cette méthodologie de programmation concilie difficilement **prototypage rapide** et efficacité des applications.

De plus, la portabilité des programmes est très limitée puisque la programmation des applications est intimement liée aux caractéristiques de l'architecture cible. Des changement mineurs sur la machine cible peuvent conduire à des temps importants de re-développement des programmes.

C'est l'objectif d'un outil d'aide à la parallélisation que de pallier ces inconvénients, en proposant d'une part un formalisme suffisamment abstrait pour l'expression du parallélisme et en automatisant d'autre part certaines tâches bas niveau, ceci afin de réduire les efforts requis respectivement par les phases de spécification et d'implantation. Le but est de permettre au programmeur de se concentrer sur les aspects algorithmiques de son problème tout en lui fournissant les moyens d'obtenir des résultats d'implantation satisfaisants en terme d'efficacité.

L'objectif de ce papier est de présenter les principes d'un outil automatique d'aide à la parallélisation en cours de développement au LASMEA. L'approche retenue pour cet outil repose sur l'encapsulation des schémas de parallélisation usuels utilisés en traitement d'images sous la forme de squelettes fonctionnels.

2 Les squelettes de parallelisation

2.1 Présentation

Au sein de notre domaine d'applications (le traitement d'images bas et moyen niveau), l'expérience accumulée par les programmeurs a montré clairement que la plupart des algorithmes implantés fait appel à un nombre limité de schémas de parallélisation, chacun admettant une ou plusieurs implantations parallèles efficaces. Par exemple, les schémas de type *décomposition géométrique - calcul - fusion des résultats* couramment utilisés pour les traitements bas niveau sont presque toujours implantés à l'aide d'un même couple topologie-harnais de communications.

A partir de là, on peut franchir un niveau d'abstraction en modélisant ces schémas sous la forme de constructeurs génériques ré-utilisables et paramétrables par les fonctions séquentielles spécifiques à une application donnée.

Formalisée par Cole[3] et Skillicorn[12], cette approche débouche sur la définition de *squelettes de parallélisation*. Un squelette est une spécification incomplète d'un schéma de parallélisation dans laquelle le programmeur va introduire ses fonctions de calculs en paramètres. Du point de vue du programmeur, l'implantation du squelette sur une plate forme parallèle est complètement cachée. Les squelettes encapsulent donc tous les aspects — communication, synchronisation, ... — relatifs à l'expression d'une forme de parallélisme. En un sens, ils sont à la programmation parallèle ce que la programmation structurée est à celle reposant sur l'utilisation des instructions *goto/label*.

Le travail de parallélisation peut dès lors se limiter au choix et à l'instantiation de ces constructeurs génériques en dehors de toute considération sur les caractéristiques physiques de l'architecture cible. Loin de restreindre l'expressivité des programmes, cette approche permet de concilier des exigences d'abstraction (sélection plutôt que ré-invention) et d'efficacité du code (l'implantation du squelette, étant faite une fois pour toute, peut être soigneusement optimisée pour une architecture donnée).

2.2 Utilisation des squelettes

Depuis leur formalisation par Cole, de nombreux auteurs ont décrit un large éventail de squelettes de parallélisation (voir [2] pour une revue). Leur introduction au niveau langage a fait l'objet de deux principales approches :

- Les squelettes sont vus comme des constructeurs parallèles spécifiques à un langage impératif séquentiel.
- Les squelettes sont exprimés comme des *Fonctions d'Ordre Supérieur* (FOS) dans un langage fonctionnel comme ML[8].

Des projets tels que P3L[1][10] utilisent la première approche. P3L est défini comme un nouveau langage structuré parallèle dans lequel les programmes sont décrits en utilisant un ensemble de primitives parallèles du langage (nommés *constructeurs parallèles*). Ces constructeurs sont des formes classiques de parallélisme telles que "pipe", "farm", etc.

Un squelette peut également être défini sous la forme d'une fonction prenant en paramètres d'autres fonctions (les fonctions séquentielles de calcul). Ce principe est largement utilisé dans les langages fonctionnels sous la forme de fonctions d'ordre supérieur. Les squelettes sont alors des objets du langages, pouvant donc être définis et manipulés à l'intérieur même de ce langage (méthode utilisée dans [9] et [4]).

Dans notre cas, cette approche permet d'associer à chaque squelette deux définitions distinctes : une définition fonctionnelle indépendante de toute architecture cible et une définition opérationnelle relative à une implantation parallèle. La première, indépendante de l'implantation, s'exprime à l'aide des FOS séquentielles du langage. Elle constitue une spécification exécutable du squelette permettant de prototyper les programmes parallèles sur plate forme séquentielle ou sur réseau de station de travail (une telle approche a été validée dans [11]). La seconde, dépendante de la cible et de la topologie utilisée, exploite les facilités offertes par l'architecture parallèle (passage de messages pour une machine MIMD à mémoire distribuée).

Cette séparation entraine de fait une plus grande portabilité des applications. Pour tout changement de la plate forme parallèle, seule la seconde définition est à reformuler, les programmes applicatifs étant inchangés.

2.3 Quelques squelettes

L'étude d'applications de vision temps réel existantes nous a conduits à sélectionner — dans un premier temps — trois squelettes fondamentaux : *SCM* (Split, Compute and Merge), *DF* (Data Farming) et *TF* (Task Farming).

1. Pisa Parallel Programming Language

2.3.1 Le squelette *SCM*

Le squelette *SCM* est utilisé pour exprimer le parallélisme géométrique. Il associe trois opérateurs (`split`, `compute` et `merge`) effectuant respectivement la partition des données d'entrée en sous domaines, le calcul sur chacun des sous-domaines et la fusion des résultats intermédiaires (voir figure 1).

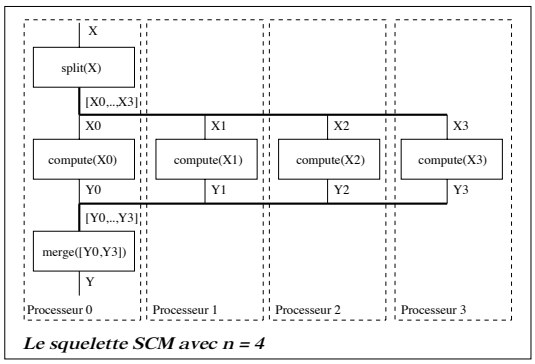


FIG. 1 – *Le squelette* SCM

Ce squelette n'est utilisable que si les données issues de chaque partition peuvent être traitées indépendamment des autres domaines. De plus, la charge de travail associée à chaque partition doit être similaire afin d'assurer un équilibre de charge sur l'ensemble des processeurs et ainsi garantir une efficacité maximale.

La définition fonctionnelle (signature et définition opérationnelle) est donnée ci dessous en Caml[2][1].

```
val scm : int -> (int -> 'a -> 'b list) ->
          ('b -> 'c) -> (int -> 'c list ->
          'd) -> 'a -> 'd

let scm n split compute merge x =
           merge n map compute (split n x)
```

avec `split`, `compute` et `merge` les fonctions séquentielles spécifiques à une application donnée, `n` le nombre de partitions et `map` est la fonctionnelle Caml permettant d'appliquer une fonction à une liste d'éléments[3]. En Caml, les types de variables sont notées : int, $'a$, $'b$ par exemple. L'expression $'a\ list$ représente le type liste de données de type $'a$. Au niveau des fonctions, l'application de f à une donnée x se

2. Caml est un dialecte de ML largement utilisé aujourd'hui

3. $map\ f[x_1, x_2, \ldots, x_n] = [fx_1, fx_2, \ldots, fx_n]$

note $f\ x$, $f\ x\ y$ s'interpréte comme $(f\ x)\ y$ ou $f(\ x,\ y)$ si la fonction admet seulement deux arguments (notation curryfiée). $f\ :'\ a-\ >'\ b$ est la fonction f de domaine 'a et de codomaine 'b.

Par exemple, une application simple utilisant le squelette *SCM* est décrite de la manière suivante :

```
par_histo = scm 4 row_block seqhisto
                  sum_histo image
```

où `row_block` est la fonction gérant la partition des images en bandes horizontales, `seqhisto` une fonction séquentielle de calcul d'histogramme et `sum_histo` somme les histogrammes calculés sur chaque bande.

2.3.2 Le squelette *DF*

Du point de vue opérationnel, la principale caractérisitique du squelette *SCM* est son comportement statique : son efficacité dépend de l'équilibre de charge des processeurs. Beaucoup d'applications utilisent des fonctions dont le temps d'exécution dépend des données. Dans ce cas, l'utilisation d'un squelette *SCM* conduit à une faible efficacité. Le squelette *DF* vise à assurer un équilibre automatique de la charge de calcul sur les processeurs de l'architecture cible. L'implantation parallèle de ce squelette est basée sur le principe de la *ferme de processeurs* dans laquelle un serveur (`Farmer`) envoie dynamiquement des données à traiter à un ensemble d'esclaves (`Worker`) et accumule les résultats intermédiaires provenant de ces mêmes esclaves.

En Caml, la définition fonctionnelle du squelette *DF* est la suivante :

```
val df : int -> ('a -> 'b ) ->
         ('c -> 'b -> 'c) ->
         'c -> 'a list -> 'c

let df n compute acc z xs =
           foldl acc z (map compute xs)
```

Ici, `n` est le nombre d'esclaves, `compute` est la fonction utilisateur de calcul, `acc` la fonction d'accumulation les résultats partiels (avec `z` la valeur initiale de l'accumulateur), `foldl` est la fonctionnelle Caml permettant d'appliquer itérativement une fonction à une liste d'éléments [4].

Sa définition opérationnelle sous la forme de quatre processus parallèles est illustrée sur la figure 2.

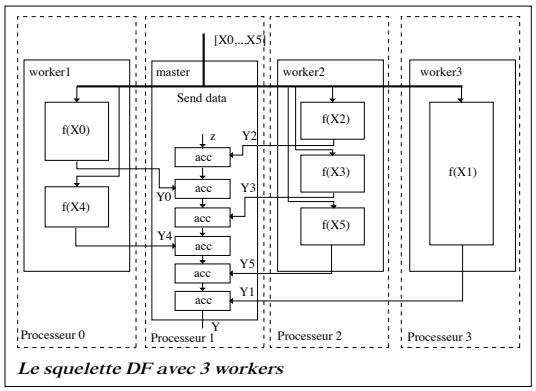


FIG. 2 – *Le squelette* DF

2.3.3 Le squelette *TF*

Le squelette *TF* peut être vu comme une généralisation du squelette *DF*. Le principe est basé sur une ferme de processeurs mais les processeurs testent chaque paquet de donnée (en utilisant un prédicat `h`). En cas de succès, la fonction `compute` est appliquée et le résultat est accumulé. Dans le cas contraire, la function `divide` est appliquée afin de générer récursivement de nouveaux paquets de données. Ce squelette correspond donc aux stratégies classiques de type "Divide and Conquer" (figure 3).

```
val tf : ('a -> bool ) ->
         ('a -> 'a list) ->
         ('b -> 'b -> 'b) -> 'b ->
         ('a -> 'b) -> 'a list -> 'b

let tf h divide combine z solve xs =
  let f x =
    if h x then
       combine z (solve x)
    else
       tf h divide combine z
       solve (divide x)
  in List.foldl combine z
     (List.map f xs)
```

4. $foldl\ f\ z\ [x_1, x_2, \ldots, x_n] = (\ \ldots(f\ (f\ z\ x_1\ )\ x_2\ )\ \ldots x_n)$

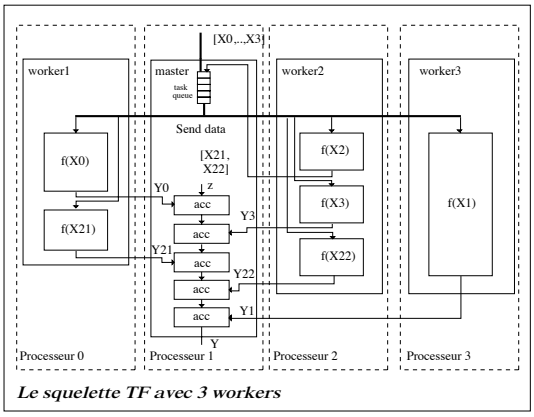


FIG. 3 – *Le squelette* TF

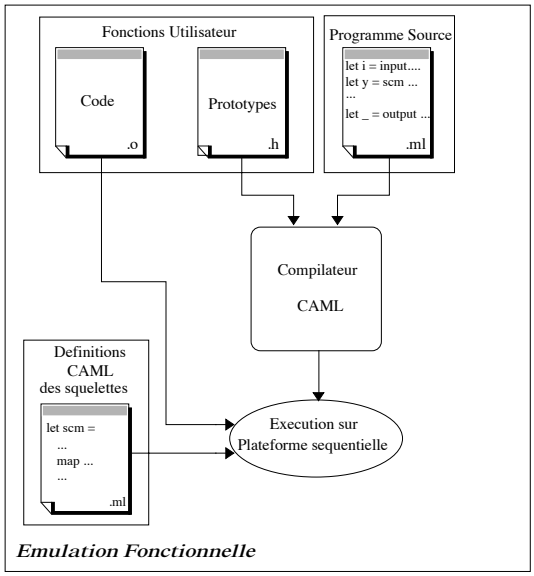


FIG. 4 – *L'émulation fonctionnelle*

3 L'outil d'aide à la parallélisation

L'outil d'aide à la parallélisation en cours de développement reprend le principe de la séparation des définitions fonctionnelle et opérationnelle des squelettes. Il est organisé autour de deux modules principaux :

- l'émulation fonctionnelle
- la génération de code parallèle

3.1 L'émulation fonctionnelle

Elle utilise les définitions des squelettes sous la forme de FOS séquentielles paramétrées par les fonctions utilisateur. Après une phase de vérification des types de données, le compilateur fonctionnel (Caml) génère directement, à partir de la spécification fonctionnelle de l'application, un code séquentiel exécutable

L'exécution de ce code sur une plate forme mono-processeur (station de travail Unix par exemple) permet de vérifier la validité algorithmique des programmes et du schéma de parallélisation employé (fig 4).

3.2 La génération de code parallèle

Cette phase se fait en deux temps :

- expansion des squelettes
- génération de code

L'expansion des squelettes — qui vise à expliciter le parallélisme contenu dans ceux-ci — repose intrinséquement sur une représentation des programmes adaptée à l'architecture cible. Dans notre cas, cette représentation est un graphe de fonctions séquentielles communiquantes (CSF). Ce modèle est similaire au modèle CSP de Hoare, la différence majeure provenant du fait que les fonctions (en opposition au processus) n'ont pas d'état interne et ne peuvent conduire à un comportement non déterministe. Chaque squelette étant développé en un graphe CSF, une application est donc un ensemble de sous graphes CSF interconnectés au sein desquels certains noeuds sont paramétrés par les fonctions de calculs associées à l'application donnée. Ce type de représentation est intéressant du fait qu'on peut manipuler et effectuer des transformations sur ce graphe, transformations visant par exemple à optimiser l'enchainement de plusieurs squelettes. Par exemple :

$$\left\{ \begin{array}{l} y = scm\ n_1\ \sigma_1\ f\ \mu_1\ x \\ z = scm\ n_2\ \sigma_2\ g\ \mu_2\ y \end{array} \right. \Longrightarrow$$
$$z = scm\ n\ \sigma_1\ (g \circ f)\ \mu_2\ x\ si$$
$$n = n_1 = n_2\ et\ \sigma_2 \circ \mu_1 = id$$

Néanmoins, utiliser un graphe CSF comme représentation intermédiaire suppose une phase de placement-ordonnancement de ce même graphe sur le réseau de processeurs. En pratique,

cette difficulté peut être surmontée en utilisant un outil existant tel que SynDEx [13]. Cet outil, déja évalué dans le cadre de travaux antérieurs [5], a été développé à l'Inria dans le cadre de recherches sur l'Adéquation Algorithme-Architecture. SynDEx est un outil d'aide à l'implantation d'algorithmes parallèles fondé sur une représentation des applications en modèle flot de données. A partir des descriptions de l'application et du réseau de processeurs, SynDEx utilise une heuristique de minimisation du temps de réponse global de l'application pour générer un placement-ordonnancement purement statique.

L'utilisation de SynDEx n'est donc possible dans notre approche que si on peut voir le graphe CSF décrivant une application comme un graphe flot de données. Pour cela, il est indispensable de respecter certaines conditions. Celles ci dépendent des squelettes utilisés et plus particulièrement du caractère dynamique ou non des communications employées au sein de ceux-ci.

En effet, SynDEx a une vision purement statique du parallélisme dans laquelle tous les aspects (allocation des variables, taille des messages à transférer,...) doivent être connus et fixés à la compilation. Dans le cas de squelettes dits "statiques" comme le squelette *SCM*, le graphe CSF associé peut facilement être transformé en graphe flot de données. Pour cela, il suffit que chaque fonction soit vue comme un opérateur purement flot de données sans état interne.

Cette interprétation n'est plus possible pour des squelettes fondés sur un schéma de communications dynamiques (comme le squelette *DF*) . En effet, pour ce type de squelettes, le nombre et l'ordonnancement des communications entre le serveur et les esclaves dépendent des données d'entrée et ne peuvent être prédites à la compilation. Ces communications dynamiques ne peuvent pas être explicitées dans le modèle d'exécutif statique utilisé par SynDEx. Elles doivent dès lors être "masquées", le graphe flot de données ne servant alors qu'à traduire les synchronisations (statiques, cette fois) de début et de fin de fonctionnement de la ferme.

Ce principe est illustré sur la figure 5.

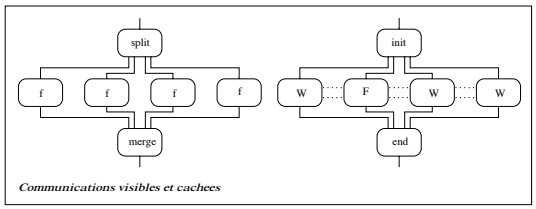


FIG. 5 – *Les communications dans les squelettes* SCM *et* DF

Les communications entre le serveur et les esclaves sont cachées dans les fonctions génériques `F` et `W`. Ces fonctions sont génériques dans le sens où, durant la phase d'implantation, elles sont construites à partir de "templates" paramétrables dans lesquels le compilateur va insérer les appels des fonctions utilisateurs et les communications dynamiques. A titre d'exemple est donnée le "template" de la fonction `W`:

```
/* start farming process */
Receive(fromFarmer,Synchro)
while(!done)
  /* Receive data */
  Receive(fromFarmer,data)
  /* no more data */
  if data == STOP then
    /* exit working loop */
    done = TRUE
  else
    /* process data using user specific
       function*/
    y=f(data)
    /* send result to Farmer */
    Send(toFarmer,y)
  endif
endwhile
/* end farming process */
Send(toFarmer,done)
```

Ce masquage des communications dynamiques impose une contrainte forte au niveau du placement-ordonnancement sous SynDEx. En effet, le temps d'exécution des fonctions effectuant ces communications étant inconnu à priori, il est obligatoire de contraindre l'heuristique de placement-ordonnancement. Pour cela, la définition opérationnelle de chaque squelette comprend pour chaque noeud du graphe flot de données le numéro du processeur sur lequel la fonction sera placée.

Le respect de ces hypothèses permet en final à SynDEx de fournir une implantation parallèle des applications. Le code généré est composé d'un fichier unique par processeur comprenant `n` séquences de communications et une séquence de calcul. Ces séquences partagent l'unique séquenceur du microprocesseur, l'ensemble étant activé par un mécanisme de synchronisation inter-séquences.

Le code généré par SynDEx (figure 6) est en réalité un macro-code intermédiaire totalement indépendant des spécificités de l'architecture cible. Son expansion en code compilable se fait à l'aide du macro-processeur `m4` d'Unix, en utilisant un ensemble de macros-définitions spécifique à l'architecture cible[5].

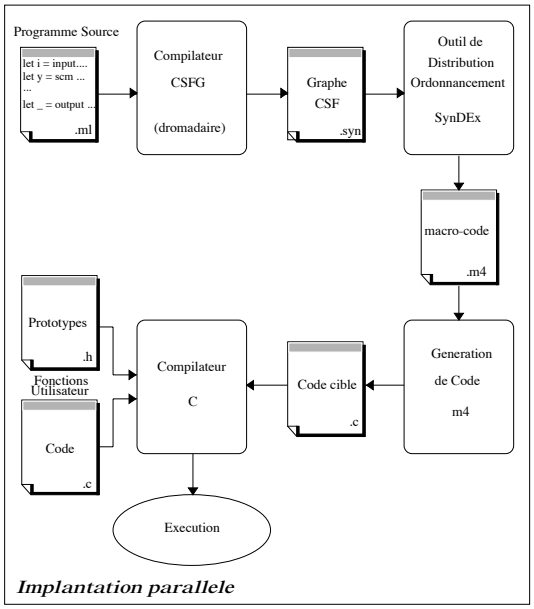


FIG. 6 – *Le générateur de code parallèle*

4 Un exemple d'application

Notre méthodologie d'implantation a été validée sur une application classique de traitement d'images : l'étiquetage en composantes connexes[6].

L'algorithme en question consiste à traiter une image dans le but de discerner les objets la composant. Pratiquement, tous les pixels appartenant à une même composante connexe se voient attribuer une et une seule étiquette. L'algorithme nécessite classiquement trois phases successives :

1. Une phase de pré-étiquetage calculant une image provisoire des étiquettes en fonction du voisinage immédiat des pixels.

2. Une phase de détection des conflits d'étiquettes, conduisant à la construction d'une *table d'équivalence*.

3. Une phase de correction réalisant la fusion des étiquettes équivalentes.

Cette application peut être décrite en utilisant deux squelettes *SCM*. Le premier calcule les étiquettes provisoires et construit une table d'équivalence globale alors que le deuxième corrige les conflits d'étiquettes à l'aide de cette même table.

De plus, du fait que l'algorithme d'étiquetage utilise en entrée une image binaire, un troisième squelette *SCM* a été utilisé afin de calculer une valeur de seuil à partir de l'image en niveaux de gris. L'algorithme utilisé, décrit dans [14], calcule un histogramme des niveaux de gris et en déduit une valeur de seuil.

La description en langage fonctionnel de l'application complète peut se faire de la manière suivante[6] :

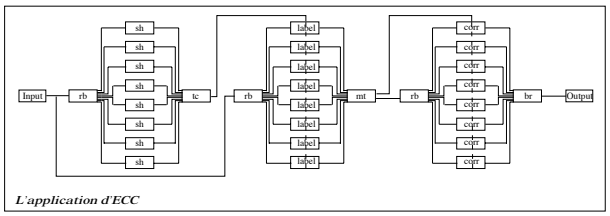
```
let image = Input 0 0 512 512 in
let threshold = scm 8 row_block seq_histo
  threshold_compute image in
let (first_label , table) = scm 8 row_block
  (label threshold) merge_tab image in
let final_label = scm 8 row_bock
  (correction table) block_row first_label in
Output final_label
```

le graphe flot de données associé à cette spécification est montré sur la figure 7.

Cette application a été implantée sur la machine Transvision sur des topologies en anneau utilisant de 1 à 8 T9000. Les résultats obtenus sont relativement médiocres. En effet, la meilleure accélération est de seulement 3.2 pour un temps d'exécution de 266 ms sur une architecture de 8 processeurs et des images de taille 512 x 512 pixels.

5. pour T800 et T9000, dans notre cas. Le développement de ces macros-définitions est en cours pour le processeur DEC 21060

6. On peut noter la concision de l'écriture de l'application



Input	get input image.
row_block (*rb*)	découpe l'image en n bandes.
seq_histo (*sh*)	calcule l'histogramme d'une bande.
threshold (*tc*)	fusionne les histogrammes et calcule la valeur du seuil.
pre-labeling (*label*)	pré-étiquetage
merge_tab (*mt*)	fusionne des tables d'équivalence locales.
correction (*corr*)	corrige les bandes d'étiquettes provisoires.
block_row (*br*)	regroupe les n bandes d'étiquettes.
Output	affiche l'image finale d'étiquettes .

FIG. 7 – *L'étiquetage en composantes connexes*

En effet, il est à noter que les performances d'une telle application sont fortement limitées par le nombre important de communications dues à l'enchainement des squelettes. Un exemple caractéristique est la complète redistribution des images d'étiquettes entre les deux derniers squelettes *SCM*.

Cette implantation peut être optimisée en constatant que la fonction de fusion `merge_tab` du deuxième squelette est séparable en deux fonctions distinctes : d'une part, une fusion des tables d'équivalence locales nécessitant peu de communications (transfert des tables et des frontières des bandes) et d'autre part, une concaténation des bandes d'images afin de reconstituer l'image complète pré-étiquetée destinée au troisième squelette. Une optimisation possible consiste à n'utiliser qu'un seul squelette *scm* effectuant entre deux opérations de calcul (pré-étiquetage et correction) une fusion interne des tables d'équivalence (figure 8).

Les résultats expérimentaux d'une telle optimisation (réalisée ici manuellement) montre une amélioration significative des performances

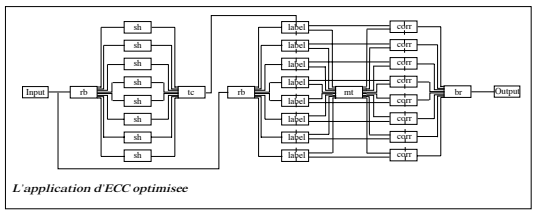


FIG. 8 – *Une optimisation*

(accélération de 4.6 et temps d'exécution égal à 186 ms).

5 Conclusion

Dans ce papier, nous avons présenté les fondements d'un outil d'aide à la parallélisation d'applications de vision artificielle sur une architecture MIMD dédiée.

L'utilisation des squelettes de parallélisation au sein de cet outil permet de dissocier les aspects fonctionnel et opérationnel des squelettes. Premièrement, la définition fonctionnelle, sous la forme de fonctions d'ordre supérieur, permet la validation des programmes parallèles sur plate forme séquentielle indépendamment de toute machine parallèle. Deuxièmement, la définition opérationnelle exploite les caractéristiques de l'architecture cible afin de fournir pour chaque squelette une implantation parallèle optimale. Cette approche a pour but de diminuer de manière drastique le temps de cycle conception-implantation des programmes, condition indispensable pour effectuer du prototypage rapide d'applications.

Les premiers résultats d'implantation, réalisés en partie manuellement sont encourageants dans la mesure où ils montrent bien l'intérêt d'un tel outil, capable de produire une version opérationnelle d'une application candidate à la parallélisation avec un minimum d'effort de la part du programmeur.

Les poursuites à court terme concernent essentiellement l'implantation d'autres squelettes et plus généralement, l'intégration des outils au sein d'un environnement de développement complet.

Références

[1] Information available online. http://pauillac.inria.fr/caml. Technical report.

[2] Duncan K.G. Campbell. Towards a Classification of Algorithmic Skeletons. Rapport de recherche YCS-276, Department of Computer Science, University of York, Décembre 1996.

[3] M. Cole. *Algorithmic skeletons: structured management of parallel computations*. Pitman/MIT Press, 1989.

[4] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe*. Springer-Verlag, June 93.

[5] D. Ginhac. Spécification et implantation d'un algorithme flots de données d'Etiquetage en Composantes Connexes sur la machine multiprocesseurs à mémoire distribuée Transvision. Mémoire de DEA d'Electronique et Systèmes, Université Blaise Pascal de Clermont-Ferrand, Juin 1995.

[6] D. Ginhac, J. Sérot, and J.P. Dérutin. Evaluation de l'outil SynDEx pour l'implantation d'un algorithme d'étiquetage en composantes connexes sur la machine Transvision. In *Journées Adéquation Algorithme Architecture*, Toulouse, 1996.

[7] P. Legrand, R. Canals, and J.P. Dérutin. Edge and region segmentation processes on the parallel vision machine Transvision. In *Computer Architecture for Machine Perception (CAMP 93)*, pages 410–420, New-Orleans, USA, Décembre 93.

[8] R. Milner, Tofte M., and Harper R. *The Definition of Standard ML*. MIT Press, 1984.

[9] N.R.Scaife, G.J.Michaelson, and A.M.Wallace. Four skeletons and a function. In *Workshop on IFL 97*, December 1996.

[10] S. Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, Universita degli studi di Pisa, Dipartimento di informatica, Mars 1993. http://www.di.unipi.it/ susanna/p3l.html.

[11] J. Sérot. Embodying parallel functional skeletons : an experimental implementation on top of MPI. In *Proceedings of Europar 97*, pages 629–633, Passau, Germany, Août 1997.

[12] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[13] Y. Sorel. Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, Mai 1994.

[14] W.H. Tsai. Moment preserving thresholding : a new approach. *Computer vision, graphics and image processing*, 29:377–393, 1985.