Vers un outil d'aide à la parallélisation fondé sur les squelettes

Dominique GINHAC, Jocelyn SEROT, Jean Pierre DERUTIN

LASMEA - URA 1793 CNRS, Campus des Cézeaux, 63177 Aubiere
e-mail: (ginhac,jserot,derutin)@lasmea.univ-bpclermont.fr

RÉSUMÉ

L'objet de ce papier est de présenter les premiers travaux relatifs à la création d'un outil semi-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

Le parallélisme reste actuellement une voie incontournable pour la conception de systèmes de vision artificielle soumis à de fortes contraintes temporelles. Toutefois, la programmation de ces architectures demeure un exercice délicat réservé à des spécialistes, entraînant de ce fait des temps de cycle conception-implantation conséquents.

Le recours à des outils d'aide à la parallélisation vise à diminuer fortement le temps d'implantation par l'automatisation de tâches bas niveau telles que la génération de code cible parallèle.

Dans ce contexte, notre approche vise le prototypage rapide d'applications de vision artificielle bas et moyen niveau sur une architecture MIMD à mémoire distribuée. Elle a pour objectif final la mise en œuvre d'un outil semi-automatique d'aide à la parallélisation, outil dédié à la machine Transvision [5] développée au LASMEA (LAboratoire des Sciences et Matériaux pour l'Electronique et d'Automatique). Un tel outil devrait permettre aux programmeurs applicatifs de porter rapidement leurs applications séquentielles et ainsi d'accroitre facilement l'expertise déjà acquise par implantation directe.

Ce papier présente les différents travaux relatifs à la création de cet outil reposant sur une description des applications de vision artificielle à l'aide de squelettes fonctionnels.

2 Présentation de l'approche

Au sein d'un domaine d'application donné, ici le traitement d'images bas et moyen niveau, l'expérience acquise par les programmeurs a révélé que les applications sont souvent construites selon un nombre restreint de schémas de parallélisation récurrents. Les schémas du type `décomposition géométrique - calcul - fusion des résultats` par exemple sont très fréquents pour les traitements bas niveau.

Vis à vis d'une architecture cible de type MIMD à mémoire distribuée, ces schémas admettent une ou plusieurs implantations parallèles efficaces. On peut alors être tenté de les abstraire sous la forme de construteurs génériques, constructeurs paramétrables par des fonctions de calcul fournies par l'utilisateur. De tels constructeurs sont appelés *squelettes*.

Le parallélisme potentiel des applications peut alors être exhibé et exploité efficacement en décrivant les algorithmes de traitement d'images sous la forme d'un enchaînement de squelettes prédéfinis.

Notre approche comporte donc quatre volets :

1. Définition d'un ensemble de squelettes appropriés au TI bas et moyen niveau.

2. Etude d'une implantation efficace pour chacun de ces squelettes vis à vis d'une topologie cible donnée.

3. Définition d'un formalisme permettant la description d'applications sous la forme d'un enchaînement de tels squelettes, paramétrés par des fonctions de partition et de calcul fournies par l'utilisateur.

4. Développement d'un environnement de programmation intégrant les points précédents et capable de fournir en sortie du code compilable pour une machine cible donnée.

3 les squelettes de parallélisation

Formalisés initialement par Cole [2], les squelettes algorithmiques sont des constructeurs génériques encapsulant les différents aspects — communication, synchronisation, instrumentation, ... — associés à l'expression d'une forme de parallélisme. En un sens, les squelettes sont à la programmation parallèle ce que la programmation structurée est à celle reposant sur l'usage de *goto/label*. En pratique, l'introduction des squelettes au sein d'un langage de programmation peut se faire de deux manières :

- soit par ajout d'annotations ou de constructeurs parallèles spécifiques à un langage impératif séquentiel, approche utilisée respectivement dans HPF [4] et le projet P3L [6] .
- soit en les exprimants directement comme des *fonctions d'ordre supérieur* (FOS) au sein d'un langage fonctionnel [7] (ML par exemple).

La deuxième approche présente de nombreux avantages, notamment le fait que la définition fonctionnelle d'un squelette sous la forme d'une FOS constitue une *spécification exécutable* de ce squelette permettant de valider les programmes parallèles sur une plateforme séquentielle avant leur implantation sur la machine cible parallèle.

3.1 Quelques squelettes

L'étude d'applications de vision temps réel existantes nous a conduit à sélectionner — dans un premier temps — deux squelettes fondamentaux : SCM (Split, Compute and Merge) et FARM (Ferme de processeurs), correspondant respectivement à l'exploitation d'un parallélisme de données régulier ou irrégulier.

3.1.1 Le squelette SCM

Le squelette SCM encapsule les schémas de parallélisation dans lesquels une même fonction (`compute`) est appliquée a un ensemble de données résultant de la partition (`split`) d'une donnée initiale, les résultats respectifs étant ensuite fusionnés (`merge`) pour donner le résultat final. Sa définition *fonctionnelle* est donnée ici (en Caml [1], un dialecte de ML) :

```
let SCM n split f merge x =
        merge n (map f (split n x))
```

Ici, `split`, `f` et `merge` sont les fonctions fournies par le programmeur, `n` est le nombre de partitions et `map` est la fonctionnelle Caml permettant d'appliquer une fonction à une liste d'éléments.

Ce squelette est typiquement utilisé pour exprimer le parallélisme "géométrique". Par exemple :

```
par_histo = SCM 4 row_block seq_histo sum_histo image
```

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

3.1.2 Les squelette FARM, DC et ITER

Le rôle du squelette FARM consiste à appliquer une même fonction `f` à une liste de données lorsque la complexité de `f` (et donc le temps de calcul associé) dépend fortement de la donnée considérée. Le but de FARM est alors d'assurer automatiquement un équilibrage de la charge de calcul sur un ensemble de processeurs. Le modèle d'implantation de ce squelette est classiquement une *ferme de processeurs*, dans laquelle un serveur distribue et collecte dynamiquement les données et les résultats au sein d'un *pool* d'esclaves.

Le squelette DC est employé pour des algorithmes nécessitant des divisions-fusions récursives (*Divide and Conquer*). Les données d'entrée sont divisées en un ensemble de sous domaines, eux mêmes partitionnés jusqu'à ce qu'ils répondent à un critère d'homogénéité.

Le squelette ITER est un squelette de haut niveau prenant en paramètre un autre squelette. Il est utilisé dès lors que les calculs d'une itération dépendent de l'itération précédente.

4 L'outil d'aide à la parallélisation

L'outil d'aide à la parallélisation, en cours de développement, rassemble tous les aspects précédemment présentés. Il s'organise autour de trois modules assurant respectivement l'émulation fonctionnelle, l'expansion des squelettes et la génération de code cible.

4.1 L'émulation fonctionnelle

L'émulation fonctionnelle (fig. 1) utilise les définitions des squelettes sous la forme de FOS. Après une phase de vérification de types (destinée à assurer la cohérence entre les fonctions utilisateur et les squelettes), le compilateur (Caml) produit un code séquentiel qui peut être utilisé pour juger de la validité fonctionnelle des résultats.

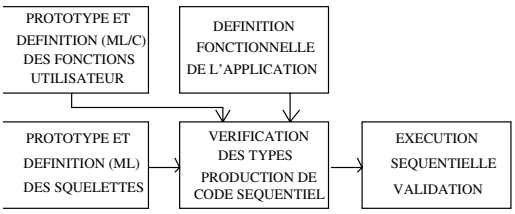

FIG. 1 — L'émulation fonctionnelle

4.2 L'expansion des squelettes

Cette étape consiste à expliciter le parallélisme exprimé par les squelettes sur la base des facilités effectivement offertes par l'architecture cible (processus concurrents, communication,...). Elle repose implicitement sur une représentation intermédiaire des squelettes (et des programmes) adaptée à cette architecture. Concernant le choix de cette représentation, deux approches ont été envisagées.

La première approche consiste à fournir une représentation des programmes sous la forme d'un graphe de squelettes

incluant les dépendances inter-squelettes, les fonctions appelées avec leur paramètres. A chaque squelette utilisé correspond alors un harnais de communication représenté sous la forme d'un ensemble de processus modèles ("templates") dans lesquels le générateur de code final insèrera les appels des fonctions utilisateurs et spécialisera les communications adéquates. Par exemple, le squelette *scm* comporte trois processus indépendants et communiquants (Split, Compute et Merge). La génération de code final est relativement simple puisqu'elle ne nécessite que trois opérations élémentaires (réservation de mémoire par création de variables, appel des fonctions utilisateur et spécialisation des fonctions de communication).

Cette voie a été testée et validée pour des applications simples ne nécessitant que peu de squelettes. Toutefois, il s'est avéré que la gestion de l'enchainement de squelettes successifs (et notamment la mise en œuvre de règles de transformation-optimisation inter-squelettes) est alors complexe.

La deuxième approche, finalement retenue, consiste à développer le graphe de squelettes en un graphe de processus communiquants (CSP), le graphe devant ensuite être placé et ordonnancé sur la topologie cible (fig. 2). Les optimisations inter-squelettes peuvent alors être définies en terme de transformation opérant directement sur ce graphe.

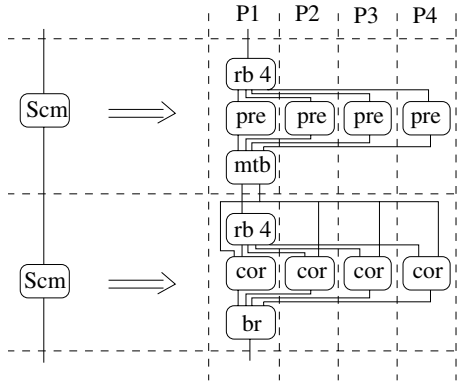

FIG. 2 — Expansion des squelettes

Concernant le placement-ordonnancement, il s'agit d'un problème classique ayant fait l'objet de nombreux travaux. Nous avons choisi de l'aborder en exploitant un outil préexistant dont nous avions l'expérience SynDEx. SynDEx [8] est un outil d'aide à l'implantation d'algorithmes parallèles développé à l'Inria dans le cadre de recherches sur l' Adéquation Algorithme-Architecture. A partir d'une spécification flot de données d'un algorithme et d'une description de la machine cible, SynDEx génère — à l'aide d'une heuristique de minimisation du temps de réponse — un placement ordonnancement purement statique. L'outil se charge par ailleurs de la génération de l'ensemble des fichiers cibles (code source de chaque processeur, makefile, ...) déchargeant ainsi le programmeur des tâches lourdes de bas niveau.

Notre travail a consisté à adopter les hypothèses requises en entrée de SynDEx à notre contexte. En particulier, le graphe CSP issu de l'expansion des squelettes doit être vu comme un graphe *flot de données* par SynDEx. Cette transformation, qui correspond à interpréter chaque processus du graphe CSP comme une fonction (sans état interne), est possible à deux conditions. Premièrement, les communications non susceptibles d'un ordonnancement statique (`farmer/worker` dans le squelette FARM par exemple) doivent être "masquées" (elles sont présentes dans le code des fonctions du graphe SynDEx mais n'apparaissent pas dans ce graphe). Secondo, l'heuristique de placement ordonnancement doit être fortement contrainte afin d'éviter les incohérences qui pourraient résulter de l'interaction des communications exprimées dans le graphe flot de données (ordonnancées statiquement) et celles qui sont "masquées" dans le code des fonctions (donc non prises en compte par cette heuristique). Pour cela, la définition opérationnelle de chaque squelette comprend, pour chaque nœud du sous graphe flot de données correspondant, outre le code de la fonction associée (auquel on ajoute les éventuelles communications "dynamiques"), le numéro du processeur sur lequel cette fonction sera placée. Moyennant ces hypothèses, SynDEx est alors en mesure de fournir un code intermédiaire sous la forme d'un macro-code indépendant de toute architecture cible. Ce macro-code se présente sous la forme d'un fichier unique par processeur comprenant n séquences de communications et une séquence de calcul partageant le séquenceur du microprocesseur, l'ensemble étant activé et désactivé par des primitives de synchronisation inter séquences.

4.3 Le générateur de code cible

Le générateur de code cible fournit le code compilable pour l'architecture cible. Il a pour point d'entrée le macro code fourni par SynDEx et est entièrement implanté à l'aide du macro-processeur *m4* d'Unix. Seule cette partie est directement dépendante de l'architecture cible. A l'heure actuelle, son portage sur une architecture mixte T9000-DEC ALPHA est en cours.

5 Premiers résultats

Les principes correspondant à la deuxième méthode ont été testés sur une application d'étiquetage en composantes connexes [3] sur une architecture multi T9000 [5]. 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 equivalentes.

Cet algorithme peut être implanté en utilisant deux squelettes *scm*, le premier étant chargé d'effectuer le pré-étiquetage et la phase de détection des équivalences, le second ayant pour tâche la correction des bandes d'étiquettes.

La description fonctionnelle de l'application peut se faire de la manière suivante :

```
let image = Input 256 256 in
let (ip,t) = scm 4 row_block pre mtb image in
let ie = scm 4 row_block (cor t) block_row ip in
Output ie
```

Le premier squelette effectue le pré-étiquetage des bandes par la fonction `pre` et la fusion conjointe des tables d'équivalence et des bandes par la fonction `mtb`. La correction des étiquettes est assurée par la fonction `cor` (paramétrée par la table d'équivalence finale *t*), la fonction de fusion des bandes corrigées étant `block_row`, fonction réciproque de l'opérateur `row_block`. Le graphe correspondant est donné figure 2.

L'implantation sur quatre processeurs a conduit à une accélération maximale de 0.83 soit 17 % inférieure à une implantation séquentielle. Ceci est dû à l'étape de réorganisation des données entre les deux squelettes *scm* (enchainement de la fusion *mtb* et de la division *row_block*).

Cette implantation peut être optimisée en constatant que la fonction de fusion *mtb* du premier 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 deuxième 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. Cette optimisation pourrait être effectuée par la première étape du générateur de code intermédiaire. Les résultats expérimentaux d'une telle optimisation (réalisée ici manuellement) montre une amélioration significative des performances (accélération de l'ordre de 1,3 à 1,6 pour quatre processeurs). Sur la figure 3, sont representées les mesures de performances d'une implantation sur quatre processeurs. Le placement des calculs et des communications sur chaque processeur est représenté sur une échelle verticale où on associe une ligne par processeur du réseau. L'échelle horizontale représente l'ordonnancement temporel à la fois des communications (Send et Receive) et des calculs (In, Split, Compute, Merge et Out). Cette figure a été générée automatiquement en utilisant une version instrumentée (avec chronométrage) des squelettes.

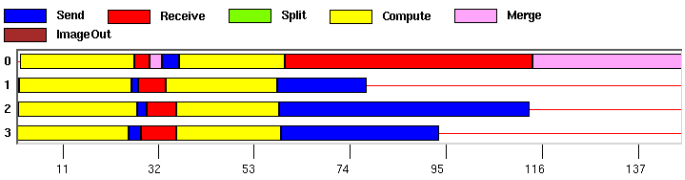

FIG. 3 — Résultats d'implantation

6 Conclusion et perspectives

Dans ce papier, nous avons présenté les fondements d'un outil d'aide à la parallélisation pour des implantations d'applications de traitements d'images sur architecture MIMD dédiée. Notre but est de diminuer de manière drastique le temps de cycle conception-implantation des applications de traitement d'images temps réel, condition indispensable pour faire du prototypage rapide d'application en vision artificielle.

Les résultats obtenus jusque là, même s'ils peuvent sembler décevants en terme de performances, sont encourageants dans la mesure où ils montrent bien l'intéret 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.

Références

[1] http ://pauillac.inria.fr/caml.

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

[3] J.P. Dérutin D. Ginhac, J. Sérot. 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.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Janvier 1993.

[5] P. Legrand J.P. Dérutin, R. Canals. 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, December 93.

[6] 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.

[7] J. Sérot. Embodying parallel functional skeletons : an experimental implementation on top of MPI. In *Proceedings of Europar 97*, 1997.

[8] Y. Sorel. Massively parallel systems with real time constraints. The "algorithm architecture adequation" methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, May 1994.