Dix-septième colloque GRETSI, Vannes, 13-17 septembre 1999

SKiPPER : un environnement de programmation parallèle fondé sur les squelettes et dédié au traitement d'images

D. Ginhac, J. Serot, J.P. Dérutin, R. Chapuis

LASMEA - UMR 6602 CNRS
24,avenue des Landais, 63 177 AUBIERE cedex, France
nom@lasmea.univ-bpclermont.fr

Résumé – Les travaux présentés dans cet article s'inscrivent dans le cadre de la méthodologie "*Adéquation Algoritme Architecture - (A^3)*". Ils concernent la conception et le développement de l'environnement de programmation paralléle SKiPPER, fondé sur les squellettes fonctionnels et permettant de faire du prototypage rapide d'applications parallèles de vision artificielle (VA) sur des architectures de type MIMD à mémoire distribuée.

Abstract – The studies which are presented in this article concern the "*Algorithm - Architecture Adequation - (A^3)*" methodology. We present a design tool based on the algorithmic skeletons for real-time vision applications aiming at significantly reducing the design-implement-validate cycle time on dedicated parallel platforms such as MIMD-DM machines.

1 Introduction

De tels outils ont pour objectif de faciliter l'évaluation rapide d'un ensemble de solutions vis-à-vis d'un problème donné en diminuant de manière drastique les temps de cycle *conception-implantation-validation* des applications. L'outil SKiPPER [3][4][5] développé dans le cadre de ces travaux est basé sur le concept des squelettes [2] de parallélisation. Ceux-ci représentent des constructeurs génériques de haut niveau encapsulant des formes communes de parallélisme tout en dissimulant les détails relatifs à l'exploitation de ce parallélisme sur la plate-forme cible. Au niveau langage, la spécification des squelettes est réalisée au sein du langage fonctionnel Caml sous la forme de fonctions d'ordre supérieur. Ainsi, la spécification d'une application est un programme purement fonctionnel dans lequel l'expression du parallélisme est limitée au choix et à l'instanciation des squelettes choisis dans une base pré-définie, chacun étant paramétré par les fonctions séquentielles de calcul spécifiques à l'application.

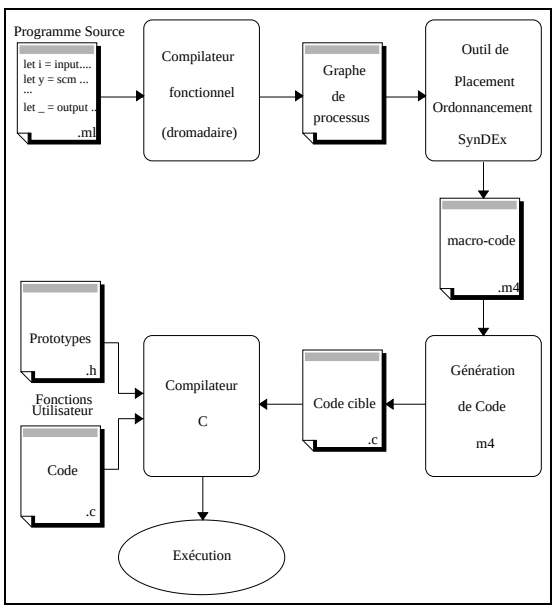

Fig. 1: L'environnement de développement SKiPPER

2 Description de SKiPPER

L'environnement de développement SKiPPER (figure 1) est organisé autour de trois modules réalisant respectivement l'expansion du code fonctionnel en un graphe flot de données (outil Dromadaire), le placement-ordonnancement de ce graphe sur l'architecture matérielle et la génération de code cible final pour l'architecture cible.

La bibliothèque de squelettes : Dans le cas général, la définition d'une bibliothèque de squelettes de parallélisation soulève un problème évident de complétude. L'originalité de notre approche a consisté à restreindre volontairement le champ d'application de tels squelettes aux seuls algorithmes de bas et moyen niveaux dans la classification usuelle en traitement de l'image. Après analyse d'un ensemble d'applications de VA parallélisées "*à la main*" au cours de nos travaux antérieurs, des principales structures algorithmiques rencontrées en VA et des principales sources de parallélisme exploitables sur structures MIMD/MD, nous avons retenu quatre classes de schémas de parallélisation :

- Les schémas dédiés au traitement géométrique des données, fondés sur un découpage en bandes de l'image à traiter.
- Les schémas dédiés à l 'extraction de caractéristiques des images. Ils sont similaires aux précédents mais nécessitent la mise en place d'une étape de fusion spécifique des résultats locaux.

- Les schémas encapsulant des structures de contrôle de type *"ferme de processeurs"* opérant soit sur des données (*data farm*), soit sur des tâches (*task farm*).
- Les schémas traduisant la nature itérative des algorithmes de vision. Ce type de schéma ne fait pas apparaitre directement de parallélisme mais permet l'implantation d'algorithme de type *prédiction vérification* travaillant par exemple sur un ensemble de fenêtres d'intérêt. Le parallélisme intervient alors lors de la mise en place, à l'intérieur de ce schéma itératif, d'un des trois schémas précédents.

A partir de ce constat, quatre squelettes élémentaires vont constituer les briques de base de notre bibliothèque :

- **SCM** (Split-Compute-Merge) regroupe les schémas des deux premières catégories.
- **DF** (Data-Farming) et **TF** (Task-Farming) représentent les structures de contrôle de type ferme de processeurs opérant respectivement sur des données et des tâches.
- **ITERMEM** (ITERate-with-MEMory) prend en compte la nature itérative des traitements sur flots d'images.

Exemples de squelettes fonctionnels : Le squelette SCM encapsule les stratégies à parallélisme de données dans lesquelles la donnée d'entrée est divisée (*Split*) en un nombre fixe de sous-domaines. Chacun des sous-domaines ainsi généré est alors traité (*Compute*) indépendamment par un processeur. Le résultat final est obtenu par combinaison (*Merge*) (selon une stratégie plus ou moins complexe) des solutions intermédiaires. La figure 2 décrit un exemple d'implantation d'un tel squelette sur une architecture à quatre processeurs. L'expression des squelettes requérant les notions de fonctions *d'ordre supérieur* (*i.e.* fonctions acceptant d'autres fonctions en argument : (les fonctions de calcul dans notre cas), et de *polymorphisme*, les fonctions de calcul séquentielles pouvant avoir une signature quelconque), le choix s'est orienté vers les *langages fonctionnels* lesquels permettent une expresion "naturelle" de ces deux notions. La définition fonctionnelle et le type du squelette SCM, décrit précédement, peuvent s'écrire par exemple sous la forme suivante en CAML :

```
> let scm n split compute merge x = merge n (pmap compute (split n x))
# val scm :
          int                     (* Nombre de partitions   *)
       -> (int -> 'a -> 'b tuple) (* Fonction de partition  *)
       -> ('b -> 'c)              (* Fonction de traitement *)
       -> (int -> 'c tuple -> 'd) (* Fonction de fusion     *)
       -> 'a                      (* Donnee                 *)
       -> 'd                      (* Resultat               *)
```

Les programmes fondés sur les squelettes passent par une phase d'expansion, le but étant de rendre explicite le comportement parallèle des squelettes utilisés. Cette phase exploite une représentation intermédiaire des programmes sous la forme de graphes de processus communicants.

La figure 3 illustre l'expansion du squelette SCM pour une parallélisation de l'algorithme de SOBEL sur 4 processeurs.

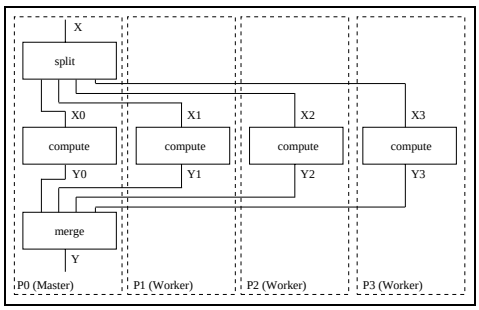

FIG. 2: Placement sur 4 processeurs

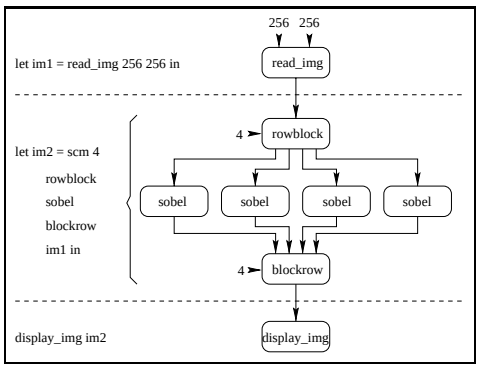

FIG. 3: Exemple d'expansion de squelettes

Placement/ordonnancement : L'étape suivante consiste à effectuer un placement et un ordonnancement de ce graphe de processus sur la machine cible. L'ensemble de ce travail s'insérant dans une collaboration avec, d'une part, le projet *SOSSO* de l'INRIA et d'autre part, le *GT7 : Adéquation Algorithme Architecture - A^3* du PRC-GDR ISIS, nous avons utilisé pour cette étape l'outil de placement/ordonnancement statique SYNDEX [7] développé par l'INRIA. A partir du graphe de processus, cet outil permet de décrire la machine cible comme un graphe de processeurs et effectue une transformation de graphe afin de faire coïncider le graphe de l'application (décrivant le parallélisme potentiel) et le graphe de l'architecture (décrivant le parallélisme disponible) tout en respectant certaines contraintes (latence minimum dans notre cas).

Génération du code cible : L'exécutif généré par SYNDEX devant être facilement portable en cas de changement d'architecture, celui-ci a été divisé en deux parties distinctes :

- un macro-code générique décrivant pour chaque processeur l'ordonnancement des opérations sous la forme d'appels de macro-définitions génériques de calcul et de communication,
- un noyau d'exécutif contenant le jeu de macro-définitions spécifiques à une architecture donnée.

La transformation du macro-code en code compilable et implantable sur l'architecture cible se fait à l'aide du macro-processeur *m4* de Unix qui remplace les appels de macro-définitions de calcul et de communication par leurs définitions spécifiques à l'architecture. Actuellement nous disposons des générateurs de code pour les processeurs Transputers T800 et T9000 équipant notre machine par-

allèle de vision *Transvision* et des travaux en cours doivent permettre d'obtenir le génération de code pour notre nouvelle machine constituée d'un ensemble de nœuds. Chacun de ces nœuds comprend un processeur de communication couplé à un processeur de calcul *DEC Alpha*

Validation de SKiPPER L'applicabilité des concepts mis en œuvre dans SKiPPER et des outils développés conjointement a été démontrée. Diverses applications de complexité réaliste : étiquetage en composantes connexes (enchainement de trois **SCM**), détection et suivi de lignes blanches en milieu autoroutier (**ITERMEM + DF**) et segmentation d'image en régions par technique de *division/fusion* (**TF**) ont été parallélisées grâce à l'environnement SKiPPER validant ainsi l'objectif initial de prototypage rapide d'applications parallèles de VA à fortes contraintes temporelles sur architecture dédiée.

3 Mise en œuvre de SKiPPER

Cette section décrit l'implantation d'un algorithme de détection et de suivi de signalisation horizontale en milieu autoroutier par vision en vue de localiser latéralement le véhicule porteur du capteur [1]. Il est basé sur une structure de type prédiction/vérification et utilise une modélisation de la route. D'une itération à l'autre, un ensemble de fenêtres d'intérêt (variable en nombre et en taille) est généré et une réactualisation du modèle de la route est réalisée par un filtrage de Kalman.

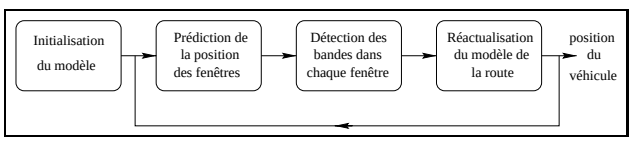

FIG. 4: Détection et suivi de signalisation autoroutière

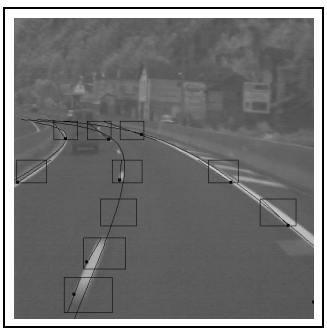

FIG. 5: Exemple de scène traitée

Description de l'algorithme Le **modèle** retenu dans [1] est basé sur un ensemble de cinq paramètres réactualisés à chaque itération (cf. figure 6) :

- C : courbure de la route considérée comme localement constante,
- x_0 : position latérale du véhicule par rapport à la bande de la route la plus à droite,
- ψ : angle de déviation horizontale de la caméra,

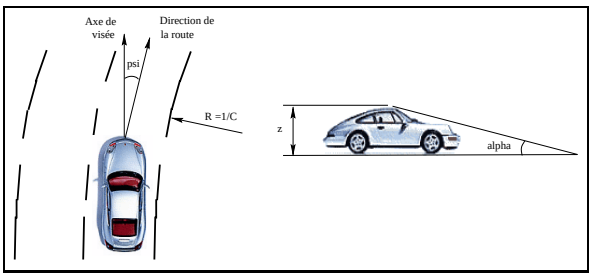

FIG. 6: Paramètres du modèle de la route

- α : angle d'inclinaison de la caméra,
- z : hauteur de la caméra.

La position estimée de la route par ce modèle permet de définir un ensemble de fenêtres d'intérêt répondant aux caractéristiques suivantes :

- le centre des fenêtres d'intérêt est placé sur la position estimée des bandes blanches,
- le nombre des fenêtres est au maximum égal à 15 (réparties uniformément sur les trois bandes de signalisation). Dans la pratique et suivant les configurations de la route, certaines de ces fenêtres sont positionnées en dehors de l'image acquise et de fait ne sont pas distribuées et traitées,
- la taille des fenêtres est fonction de leur position dans l'image. En effet, les zones d'analyse placées dans le haut de l'image c'est-à-dire vers l'horizon sont de taille inférieure à celle positionnées dans le bas de l'image pour ne pas risquer d'inclure plusieurs bandes dans une même fenêtre et ainsi fausser les résultats.

La **phase de détection** a pour objectif l'extraction des bandes de signalisation dans chaque fenêtre d'intérêt. Pour cela, ces bandes sont assimilées à deux segments parallèles. Leur détection fait appel à un calcul de gradient horizontal suivi d'une recherche des segments par transformée de Hough.

La **phase de réactualisation du modèle** est confiée à un filtre de Kalman qui, à partir des résultats de détections dans chaque fenêtre d'intérêt, met à jour les trois paramètres du modèle de la route.

Specification fonctionnelle : Premièrement, l'application est basée sur une stratégie de calcul opérant sur un ensemble de fenêtres d'intérêt sur lesquelles on applique successivement différents traitements (seuillage, gradient horizontal et transformée de Hough). Etant donné que ces traitements sont indépendants d'une fenêtre à l'autre, il est possible d'utiliser un schéma de parallélisation encapsulant une stratégie à parallélisme de données. Un simple partage de données (utilisé dans le squelette SCM) n'est pas préconisé du fait que la taille et le nombre des fenêtres d'intérêt n'est pas figé et évolue en fonction des configurations de la route. L'utilisation d'un squelette SCM pourrait alors entraîner un important déséquilibre de charge conduisant à une faible efficacité des implantations résultantes. Pour obtenir une répartition équilibrée sur l'ensemble des processeurs, il est nécessaire de distribuer les traitements de manière dynamique en fonction

des besoins de l'application d'où le recours à un squelette de type DF.

Deuxièmement, la stratégie de type prédiction-vérification fait apparaître explicitement la notion de flot continu d'information : les traitements à l'itération i se terminent par la réactualisation du modèle de la route qui sert de point d'entrée de la phase de distribution des fenêtres d'intérêt. La mise en œuvre de ce "rebouclage" nécessite donc l'utilisation d'un squelette ITERMEM.

Ainsi la spécification fonctionnelle de l'application de détection et de suivi de lignes blanches ainsi que les prototypes des fonctions applications peuvent être écrites par l'utilisateur de la manière suivante :

```
let x0          = init 0 in              (* Initialisation du systeme  *)
let f (x,i)     =                        (* Definition de fonction     *)
  let fenetres  = prediction x i in      (* Phase de prediction        *)
  let x'        = df nbproc
                      detection          (* Phase de detection         *)
                      maj                (* Accumulation de detections *)
                      x                  (* Accumulateur initial       *)
                      fenetres in        (* Liste de fenetres          *)
  evolution x' in                        (* Reactualisation du modele  *)

itermem read_img                         (* Lecture d'image            *)
        f                                (* Appel de la fonction f     *)
        affiche                          (* Affichage                  *)
        x0                               (* Etat initial               *)
        (512,512)                        (* Taille de l'image          *)
```

```
void read_image(int nrow, int ncol, image *out);
void init(int val, etat *x0);
void prediction(etat xc, image in, fenetreList* fs);
void detection(fenetre fen, detect *out);
void maj(etat xc, detect r, etat* xs);
void evolution(etat xc,  etat* xs,  etat *xc);
void affiche(etat xc);
```

Dans l'étape suivante, le compilateur fonctionnel *dromadaire* génére le graphe flot de données représenté sur la figure 7 dans le cas où le nombre de processeurs esclaves est égal à 4, ce dernier étant le point d'entrée de l'outil SYNDEX de placement-ordonnancement.

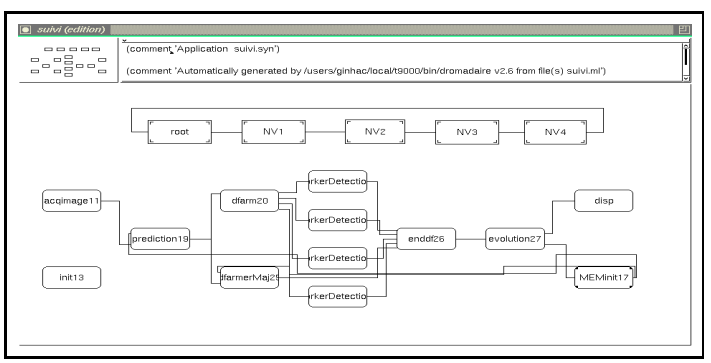

FIG. 7: Graphe flot de données de l'application de détection de bandes blanches

4 Résultats de prototypage rapide

Deux types de résultats sont montrés :

- l'évaluation des temps d'exécution sur la machine cible TRANSVISION [6] en fonction d'une topologie en anneau de 2 à 8 processeurs de type transputer T9000. Ceci montre l'intérêt d'un tel outil lors d'une extension du nombre de processeurs : ce nombre étant un simple paramêtre du squelette DF. Le meilleur temps obtenu est de 50 ms de temps par boucle d'itération avec 8 processeurs.
- l'évaluation en coûts de développement lesquels se traduisent par :
  - une spécification fonctionnelle de quelques lignes en langage `CAML`,
  - un temps de développement de 3 jours/homme,
  - 1000 lignes de codes `C` développées par l'algorithmicien pour 6000 lignes de `C` parallèle générées.

5 Conclusion

Cet article permet de présenter l'environnement de programmation parallèle SKiPPER et sa validation par le prototypage rapide d'un algorithme de vision réaliste à fortes contraintes temporelles. Cette démarche s'inscrit pleinement dans la problématique scientifique *Adéquation Algorithme Architecture*. De nouveaux travaux vont porter sur la spécification et l'implantation de règles de transformations inter-squelettes afin de décrire des applications utilisant des enchainements complexes de squelettes de parallélisation. Ces règles ont pour objectif d'optimiser le graphe de processus issu de cet enchainement. Dans ce contexte, les relations entre squelettes et le concept de granularité (de tâches et de données), et les relations entre exécutifs statique et dynamique seront étudiées. La diffusion de l'outil SKiPPER dans la communauté scientifique fait partie de nos objectifs à court terme.

References

[1] R. Chapuis. *Suivi de primitives image, application à la conduite automatique sur route*. PhD thesis, Université Blaise Pascal, Jan 1991.

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

[3] D. Ginahc. *Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels*. PhD thesis, Université Blaise Pascal - Clermont 2, Janvier 1999.

[4] D. Ginhac, J. Sérot, and J.P. Dérutin. Vers un outil d'aide a la parallélisation fondé sur les squelettes. In *16^{eme} Colloque GRETSI sur le traitement du signal et des images*, Sep, 15–19 1997.

[5] D. Ginhac, J. Sérot, and J.P. Dérutin. Utilisation de squelettes fonctionnels au sein d'un outil d'aide à la parallélisation. In *4^{emes} Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, Jan, 28–30 1998.

[6] 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*, pages 410–420, New-Orleans, USA, Dec, 15–17 1993.

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