EADS **Aerospatiale Matra Missiles**	**RAPPORT DE STAGE**	Classification du document N C

Renseignements concernant le stage :

Sujet : Prototypage rapide d'une application de traitement d'images avec SynDEx

Dates : du 09 octobre 2000 au 31 août 2001

Unité ou service d'accueil : E/SCS/V

Maître de stage : Christophe SCHLOSSERS

Renseignements concernant le stagiaire :

Nom : Kamel MEZIANE

Ecole : Université Paris XII, Val de Marne

Niveau d'études: D.E.S.S.

Spécialité: E.E.A. - Option Signaux, Images et Systèmes temps réel

Responsables de stage de l'école: Eric DELECHELLE – Eric PETIT

Diffusion du rapport:

Circuit original	Maître de stage	Agent de sécurité	Sce.Emploi Formation	Stagiaire	Maître de Stage
	● - - - ➤	● - - - ➤	● - - - ➤	● - - - ➤	●
	Visa - Date	Visa - Date	Visa - Date	Date et lieu de soutenance	Archivage
Circuit Copie				Responsable de stage de l'école	Sce.Emploi Formation
				● - - - ➤	●

TABLE DES MATIERES

TABLE DES FIGURES

Présentation du stage :

Ce stage s'inscrit dans le cadre du projet ACOTRIS (Analyse et Conception à Objet Temps Réel pour Implantation asynchrone/Synchrone) du programme RNTL (Réseau National des Technologies Logicielles) auquel participe : Aérospatiale Matra Missiles (AMM), le CEA-Leti, CS SI, l'INRIA/IRISA et la société SITIA.

ACOTRIS se propose d'aider à la conception d'applications temps réel embarquées imposant un niveau de parallélisme important en nous basant sur un support de conception UML et les outils de génération, dimensionnement et placement des applications issues des approches synchrones, ou plus largement multi-horloges.

Dans ce projet, l'Aérospatiale Matra Missiles se doit de :

- réaliser une application de référence de traitement d'images,
- réaliser une application projet de traitement d'images,
- valider la démarche retenue pour le traitement d'images.

Le travail a consisté à développer avec SynDEx (outil développé par l'INRIA) l'application de référence indépendamment de l'approche ACOTRIS/UML-Signal-SynDEx.

Résumé :

L'objet du présent rapport est l'étude du prototypage d'une application de traitement d'images avec SynDEx (**Syn**chronized **D**istributed **Ex**ecutive).

Dans un premier temps, nous présenterons l'application de calcul d'une translation entre deux images.
Nous introduirons, ensuite, la spécification en flots de données ainsi que les principes de SynDEx.
Nous effectuerons l'implantation de notre algorithme de calcul d'une translation entre deux images avec SynDEx v5 (sans parallélisme de données) puis avec SynDEx v6 (avec du parallélisme de données).
Nous explorerons, enfin, différentes implantations de notre algorithme de calcul d'une translation entre deux images sur différentes architectures afin de déterminer l'implantation optimisée de notre algorithme.

Mots-clé :

ACOTRIS, prototypage rapide, calcul d'une translation entre deux images, ADSP21060, graphe flot de données, parallélisme potentiel, implantation optimisée, SynDEx, Adéquation Algorithme Architecture,.

1 Présentation de la société et du service.

1.1 Aérospatiale Matra Missiles.

Le groupe Aérospatiale est né le 1er janvier 1970 de la fusion de trois sociétés : Sud Aviation, Nord Aviation et Sereb sous le nom de SNIAS (Société Nationale Industrielle Aérospatiale). En 1984, la SNIAS prend officiellement le nom d' «AEROSPATIALE ».

Depuis le 1er avril 1999, la division Missiles est devenue une société à part entière : Aérospatiale Missiles puis Aérospatiale Matra Missiles suite à la fusion, le 11 juin 1999, de Aérospatiale et de Matra Hautes Technologies créant ainsi le groupe AEROSPATIALE MATRA. Elle était une filiale à 100% du Groupe Aérospatiale Matra et a intégré EADS depuis juillet 2000.

Bientôt elle fera partie de la Société Européenne de Missiles qui sera constituée de :

- Aérospatiale Matra Missiles avec 2962 personnes ;
- Matra BAe Dynamics (MBD) avec 5039 personnes ;
- La division Missiles d'Alenia Marconi Systems avec entre 1419 et 1869 personnes.

Celle-ci s'adossera sur les grands groupes suivants :

- EADS actionnaire indirect à 37.5% ;
- Bae Systems actionnaire indirect à 37.5% ;
- Finmeccanica actionnaire indirect à 25%.

Cette politique de regroupement vise à constituer un ensemble leader mondial dans son domaine permettant ainsi d'organiser et de consolider une grande industrie européenne aéronautique et de défense.

1.2 Le service de Vision.

La croissance rapide des besoins en Vision et Traitement d'Images dans les systèmes d'armes a conduit Aérospatiale Matra Missiles à développer une forte compétence dans ce domaine depuis 1988.

Le service actuel regroupe 30 personnes spécialisées dans les différentes disciplines du traitement du signal et de l'image, la modélisation de scènes dans différentes longueurs d'onde (visible, infrarouge et radar) et l'architecture fonctionnelle des chaînes de vision. Il s'appuie sur les compétences des services équipements optroniques et radar, pour la modélisation de la scène et des capteurs, et des secteurs guidage pilotage et navigation pour la modélisation du comportement des engins. Le service travaille en coopération étroite avec le département électronique pour la réalisation des ASICs (Application Specific Integrated Circuits), cartes et calculateurs temps réel.

a. Missions du service

La mission du secteur "Vision" est de mener à bien les études et développements nécessaires à la définition, la conception et la mise au point des équipements et systèmes à imagerie dans les différents programmes d'Aérospatiale Matra Missiles. La méthodologie mise en place s'appuie sur la modélisation et la simulation des différents composants de la chaîne fonctionnelle de Vision des systèmes d'armes (terrain, atmosphère, plate-forme, capteur, traitements de bas et haut niveau) en tenant compte, suivant le système considéré, de l'implication de l'homme dans la boucle.

Pour les applications Traitements d'Images Temps réel, le rôle du service est de concevoir, développer et valider les chaînes d'algorithmes et de les maquetter sur plate-forme temps réel, afin d'aboutir à un démonstrateur permettant d'en évaluer les performances puis de spécifier un calculateur opérationnel. Le maquettage temps réel nécessite la définition de l'architecture du calculateur et le développement du logiciel temps réel. Pour tenir les performances nécessaires, il est parfois nécessaire de concevoir, de spécifier et de faire réaliser les cartes et les ASICs.

Le secteur est titulaire de contrats d'études spécialisées sur les différents systèmes d'armes Aérospatiale Matra Missiles (Anti-Char, Sol-Air, Sol-Sol, Anti-Navire multi-rôle, Renseignement,...). Il est donc impliqué dans la maîtrise des grandes fonctions de la Vision :

- **Détection – Reconnaissance - Identification** d'objectifs; maquette temps réel en contexte multi-spectral (visible, infrarouge bande 2 et bande 3) et multi-contexte (Sol-Air, Sol-Sol et Anti-Navire).
- **Localisation et poursuite de cibles :** veille, poursuite de cible et fusion multicapteur pour les Sol-Air, poursuite pour Anti-Char, balise intelligente de surveillance.
- **Préparation de mission, recalage de navigation et guidage terminal** d'engins autonomes; maquettes temps réel pour du guidage infrarouge. Spécification et recette d'ASICs.
- **Visualisation et aide à l'opérateur**; maquettes temps réel pour Sol-Sol, compression-transmission intelligente d'images, détection-comptage d'objets dans des séquences d'images très haute cadence.
- **Manipulation et enrichissement de données cartographiques** et images multi-sources pour le renseignement, l'aide au commandement et les drones (station sol de préparation, de suivi et d'exploitation de mission).

b. Organisation

Ces différents travaux ont conduit à structurer les compétences selon 7 axes d'activités:

- **Simulation des chaînes de vision**:
  Développement et amélioration des méthodes de conception, des outils de développement et des procédures d'exploitation et de validation pour l'allocation de performances (du système et de ses composants) ; puis suivi des développements.

- **Analyse d'images:**
  Temps réfléchi :
  Développement et amélioration des algorithmes de reconstruction et caractérisation automatisée d'objets dans la scène par imagerie (application en particulier à la cartographie et au renseignement),

Temps réel :
Développement et amélioration des algorithmes de stabilisation, localisation, détection reconnaissance et poursuite d'objets, transmission/compression, souvent dans des séquences d'images. La prise en compte de la contrainte temps réel nécessite l'adaptation algorithme-architecture matérielle pour le développement de calculateurs de traitement d'images dédiés.

- **Fusion de données multicapteurs**:
  Développement et amélioration des méthodes et algorithmes de fusion. La fusion de données se décline en fusion d'images entre elles au niveau pixel pour faciliter l'interprétation, en fusion d'attributs extraits des images pour préparer une décision et optimiser le réglage des capteurs et en fusion d'images et de données hétérogènes en particulier des données cartographiques pour la préparation de mission, l'observation aérienne ou du champ de bataille.

- **Architecture Temps Réel**:
  Réalisation de démonstrateurs de traitement d'images temps réel. L'objectif est de prendre en compte au plus tôt les contraintes associées au fonctionnement en temps réel des algorithmes de traitement d'images afin d'optimiser le développement. L'approche permet de définir rapidement une architecture logicielle et matérielle permettant de valider en temps réel les performances, puis de spécifier un calculateur de traitement d'images temps réel embarqué, réalisé ensuite par le secteur spécialisé d'Aérospatiale Matra Missiles.

- **Synthèse d'images**:
  Modélisation multispectrale (paysages et cibles, visible, infrarouge et radar) et reconstitution temporelle de la scène (scénarios, animation).

- **Qualité image**:
  Définition des paramètres de qualité pour les systèmes à imagerie terrestres, marins, aériens ou spatiaux ; définition et constitution des outils de recette de la qualité géométrique et radiométrique des images.

- **Cartographie et élaboration de données**:
  Approvisionnement et évaluation des différentes sources de données disponibles pour la préparation des missions des engins (données images, cartes ou fichiers vecteurs). Traitements pour générer les cartes prévisionnelles 2D et les modèles 3D embarqués par les engins autonomes par exemple. Elaboration des bases de données scènes pour l'activité synthèse d'images. La compétence en géographie permet de rechercher et de définir les caractéristiques terrains des domaines d'emploi des systèmes d'armes.

2 Algorithme séquentiel de calcul de translation entre deux images

2.1 Contenu de la livraison

Début décembre 2000, le document enregistré sous le n°E/SCS-2000-759-A (fourni en Annexe 1), composé d'une trentaine de pages, m'a été remis.

Il décrit le noyau de l'algorithme de calcul d'une translation entre deux images qui servira de Spécification Technique de Besoins pour les applications Référence et Projet, et donne le synoptique de l'architecture matérielle de la carte sur laquelle sera implanté l'algorithme. Ce synoptique servira de base pour réaliser le modèle d'architecture.

L'algorithme (fichiers sources et d'en tête) est programmé en langage C.

2.2 Étude fonctionnelle

L'algorithme est fondé sur une succession de calculs de translation en repère image entre deux images successives d'une séquence d'images. Il utilise des points caractéristiques extraits par chacun des ASICs de l'architecture sur chacune des images. L'appariement des points caractéristiques homologues s'effectue par corrélation locale inter-image. Une translation en x et y est ensuite calculée entre les deux images.

Brièvement cet algorithme est constitué par l'enchaînement des fonctions suivantes :

1. Pré-estimation de la translation globale entre les images

Nous commençons par masquer les deux images (nous ne gardons que l'octet de poids faible de chaque pixel de l'image d'origine). Ensuite, nous réduisons les images d'un facteur 2 pour diminuer le nombre de calculs, puis nous effectuons une corrélation sur toute l'image 2 avec un masque (imagette) extrait du centre de l'image 1. Ceci fournit une translation estimée globale.

2. Appariement des points caractéristiques

Pour tous les points caractéristiques extraits de l'image 1

Pour tous les points caractéristiques extraits de l'image 2

Nous effectuons une première corrélation simple sans déplacement autour de la position du point caractéristique pour obtenir un score indiquant si les deux points peuvent être appariés

Si le score est bon :

Corrélation fine avec estimation de la position précise du point caractéristique, et calcul du score.
Nous gardons le point avec le score minimum.

Fin des points de l'image 2
Calcul de la confiance pour le point avec le score minimum
Vérification si pas de conflit d'unicité

Fin des points de l'image 1

3. Filtrage des appariements incorrects

Nous calculons le vecteur de translation pour chaque point, et nous considérons qu'un appariement est correct si un pourcentage minimum de vecteurs de translation des autres appariements est compatible avec lui.

4. Calcul de la translation fine

Après l'appariement nous avons deux listes de points (représentant les points d'origine et les points après la translation) et le nombre (N) de points constituant les deux listes. Pour calculer la translation fine nous commençons par prendre la première translation que nous appliquons aux (N-1) autres points d'origine. Ensuite nous prenons la seconde translation que nous appliquons aux (N-1) autres points d'origine. Nous gardons comme résultat la médiane des erreurs pour cette translation. Nous renouvelons ces calculs pour le reste des points de la liste. Pour chacun des N déplacements associés aux N points, nous calculons la médiane des N-1 erreurs et nous obtenons à la fin la médiane des erreurs de chaque translation (n translations). Nous gardons à présent le minimum de la médiane des erreurs. La translation de l'image correspond à la translation ayant le minimum des N erreurs médianes.

Le graphe flot de contrôle de cet algorithme est représenté figure 1. Une première étude du parallélisme de l'algorithme nous permet d'obtenir un graphe (Fig. 2) faisant apparaître du parallélisme.

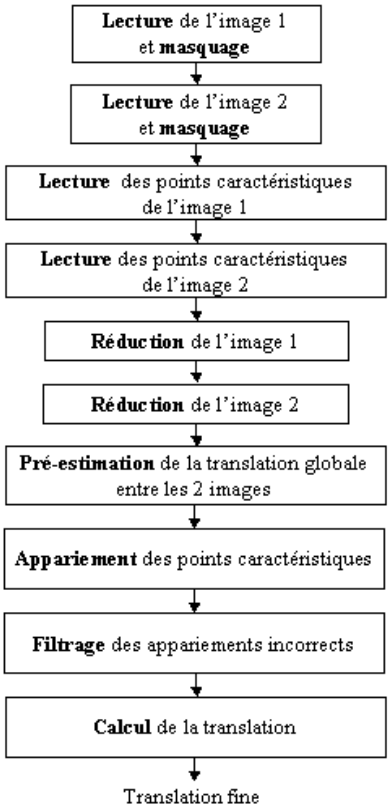


Figure 1 : Graphe flot de contrôle de l'algorithme.

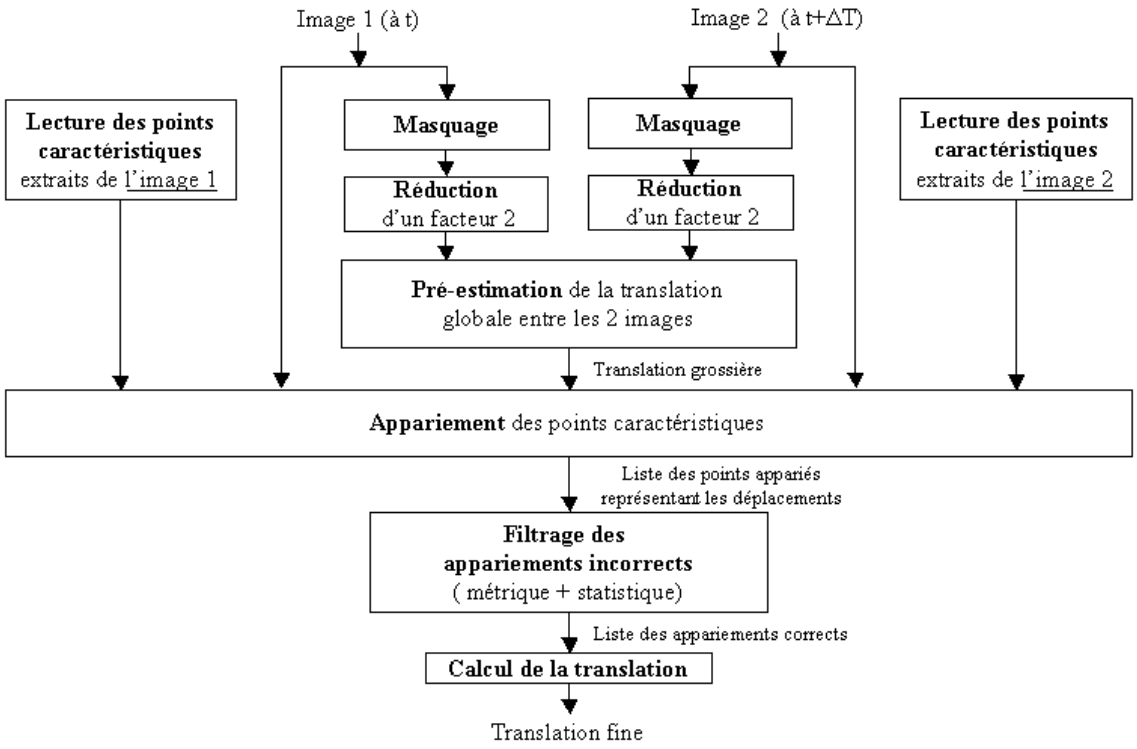


Figure 2 : Graphe faisant apparaître un peu de parallélisme.

2.3 Fichiers d'entrées/sorties de référence

La livraison logicielle de l'algorithme (fichiers de programmes « .c », « .h », deux fichiers images et deux fichiers de points caractéristiques) a été faite le 17/01/2001. Pour les besoins de la simulation, les images et les listes de points caractéristiques extraits, sont sous forme de fichiers.

Pour vérifier la bonne compréhension de l'algorithme ainsi que son bon fonctionnement, nous testons l'application en séquentiel en exécutant le programme C sur une station de travail afin de constituer les fichiers de sorties référence. Ceux-ci seront notre élément de comparaison lors de l'implantation de l'algorithme avec SynDEx.

3 Spécification parallèle de l'algorithme

3.1 Spécification en flots de données

Il existe deux approches principales pour spécifier un algorithme : celle flot de contrôle et celle flot de données.

Dans un graphe flot de contrôle, qui est le modèle de graphe correspondant aux langages impératifs tels que C, Fortran, etc.…, chaque sommet représente une opération qui consomme et produit des données dans des variables pendant son exécution, et chaque arc représente une précédence d'exécution sur ces opérations. L'ensemble des arcs définit un ordre total d'exécution sur les opérations. Il n'y a pas de relation, entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données dans les variables.
Dans un graphe flot de données qui est le modèle de graphe correspondant aux langages flot de données, chaque sommet représente une opération qui consomme toutes ses données d'entrées avant son exécution et produit toutes ses données de sorties après son exécution, introduisant ainsi un ordre entre la lecture des données sur toutes les entrées et l'écriture des données résultats sur toutes les sorties. La notion de variable n'existe pas au niveau de la spécification d'un algorithme. L'ensemble des arcs définit un ordre partiel d'exécution sur les opérations traduisant naturellement du parallélisme potentiel de tâches (opérations différentes appliquées à des données différentes) ou de parallélisme potentiel de données (même opération appliquée à des données différentes). Il y a maintenant une relation, entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données. Par ailleurs, le graphe flot de données spécifiant une application temps réel correspond à la répétition infinie (temporelle si nous avons un seul capteur ou/et spatiale si nous avons plusieurs capteurs) d'un motif de graphe qui s'exécutera à chaque acquisition par le ou les capteurs, des données venant de l'extérieur. Nous pouvons décrire la répétition finie d'une opération ou d'un graphe d'opérations de la même manière que pour la répétition infinie. Si cette répétition est spatiale cela correspond à du parallélisme potentiel de données. Enfin, il est facile de transformer, si cela est nécessaire lors des optimisations, une répétition spatiale en une répétition temporelle (correspondant à une itération) et vice-versa.

Nous pouvons faire apparaître du parallélisme potentiel dans un graphe flot de contrôle qui n'en possède pas. Nous devons pour cela le transformer en un graphe flot de données par analyse des dépendances de données entre opérations à travers l'accès aux variables.
Notre algorithme est écrit en langage C, et il est représentable par un graphe flot de contrôle (Fig. 2). Nous voulons faire apparaître le maximum de parallélisme potentiel, pour cela il faut décrire notre algorithme en flot de données. Le passage de la spécification flot de contrôle à flot de données nécessite une étude plus approfondie de l'algorithme que l'étude fonctionnelle. Cette étape est très importante, car c'est de cette étude que nous déduisons le parallélisme potentiel de l'algorithme.
Tout d'abord, il faut adapter les fonctions du programme C en supprimant les variables globales pour les rendre locales à la fonction et réécrire cette fonction afin de lui donner la structure : lecture des entrées, calculs utilisant toutes les entrées, écriture des sorties. Ensuite, nous analysons les dépendances de données entre les différentes fonctions. S'il n'y a pas de dépendances de données entre deux fonctions, cela veut dire qu'elles sont potentiellement parallélisables, dans le sens où elles seront mises effectivement en parallèle si et seulement si les ressources matérielles le permettent. Chaque fonction réécrite de cette manière correspond à une opération (sommet) du graphe de l'algorithme que l'on cherche à obtenir. Lorsqu'une fonction doit transmettre une donnée à une autre fonction cela se traduit par un arc entre deux opérations. L'ensemble des arcs définit un ordre partiel d'exécution sur les opérations. Enfin, nous devons transformer les boucles ou itérations (répétitions temporelles) en répétitions

spatiales d'opérations ou de sous-graphes d'opérations, ce qui fait apparaître du parallélisme potentiel de données. Chaque boucle est spécifiée par un couple de sommets particuliers « F » (Fork), « J » (Join) respectivement entrée et sortie du graphe à répéter spatialement. Nous pouvons aussi utiliser un sommet particulier « D » (Diffuse) pour diffuser spatialement une donnée et un sommet particulier « I » (Iterate) pour décrire des dépendances inter-répétition.

La figure 3 présente un exemple de graphe d'algorithme effectuant un *produit matrice-vecteur récursif* correspondant à la répétition infinie d'un produit matrice-vecteur (MV) dont le vecteur à 3 éléments e_t est fourni à chaque acquisition par un capteur et la matrice 3x3 s_t est fournie via un retard $ (dépendance inter-répétition infinie) par le résultat du produit matrice-vecteur zs_{t-1} calculé lors de la répétition précédente *t-1*. Le résultat de chaque produit matrice-vecteur est aussi transmis (diffusé) à un actionneur. La figure 4 présente le sous-graphe qui calcule un des produits matrice-vecteur correspondant à trois répétitions du produit scalaire V. La figure 5 présente le sous-graphe qui effectue un produit scalaire correspondant à trois répétitions des opérations multiplication-accumulation. Ce sous-graphe utilise trois dépendances inter-répétitions finies pour effectuer l'accumulation à partir du résultat de la somme calculée à la répétition précédente.

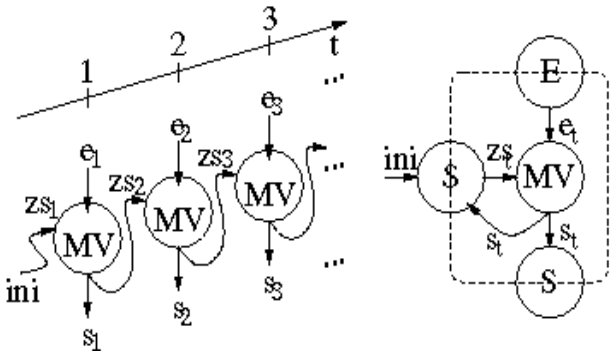


Figure 3 : Graphe d'algorithme (répétition infinie du produit matrice-vecteur)

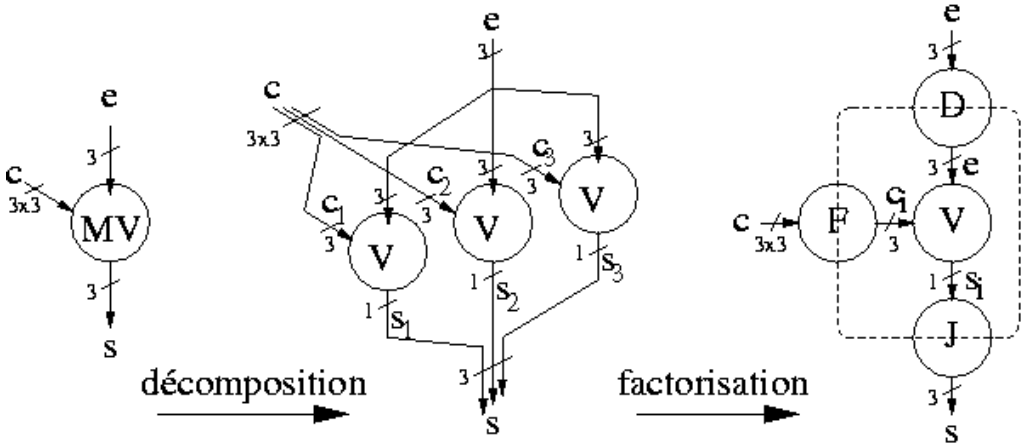


Figure 4 : Sous-graphe MV (3 répétitions finies de V)

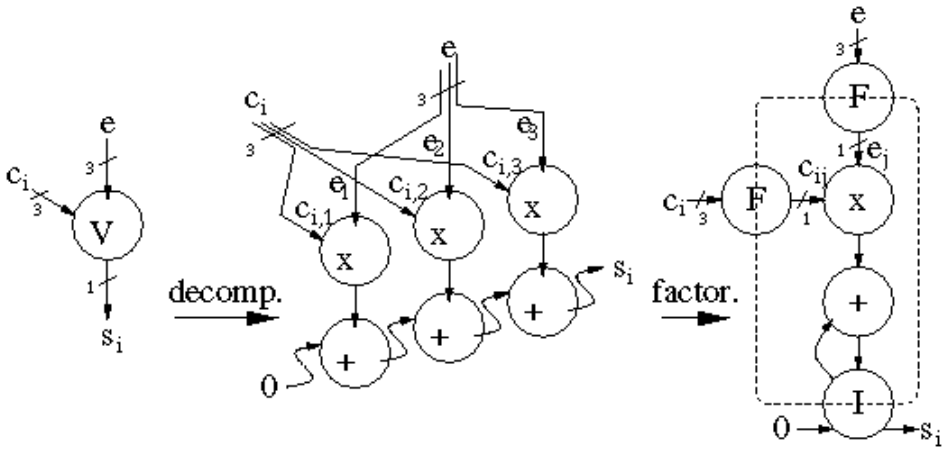


Figure 5 : Sous-graphe V (3 répétitions finies de multiplication-accumulation)

Nous avons appliqué cette méthode sur notre programme C de calcul de translation entre deux images, ce qui a conduit au graphe flot de données (utilisable avec SynDEx v6) dont un extrait est représenté sur la figure 6.

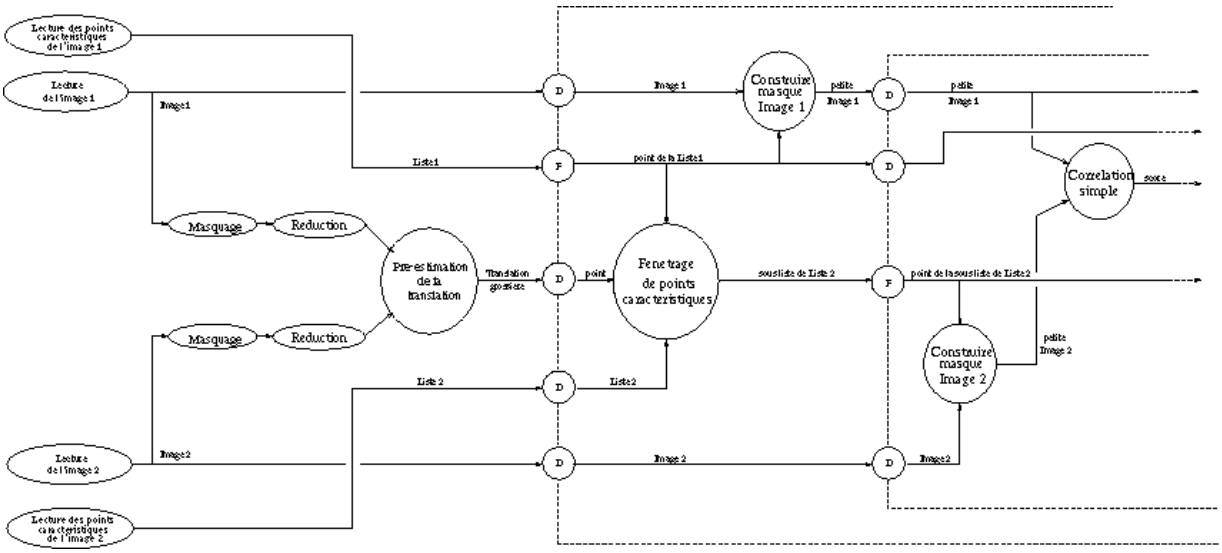


Figure 6 : Début du graphe flot de données de l'algorithme de calcul de translation.

3.2 Spécification simplifiée pour SynDEx v5

Il est fastidieux avec SynDEx v5 de faire apparaître du parallélisme de données (même opération répétée sur des données différentes) car il faut exprimer explicitement les répétitions d'opérations ou de sous-graphes d'opérations (ce qui n'est pas le cas pour SynDEx v6). Nous réalisons donc à partir de la spécification flot de données (extrait, Fig. 6) une spécification simplifiée (Fig. 7) dans laquelle nous encapsulons dans une unique opération les opérations répétées, délimitées par les sommets particuliers (« F », « J », « D » et « I ») que nous remplaçons par les ports d'entrées ou de sorties de cette opération. Nous obtenons donc un graphe d'algorithme constitué de la lecture dans quatre fichiers des images 1 et 2 et de leurs points caractéristiques, du masquage et de la réduction sur les deux images, de la pré-estimation, de l'appariement des points caractéristiques qui correspond à l'opération encapsulée vue précédemment, du filtrage des appariements incorrects, et enfin de l'écriture des appariements dans un fichier.

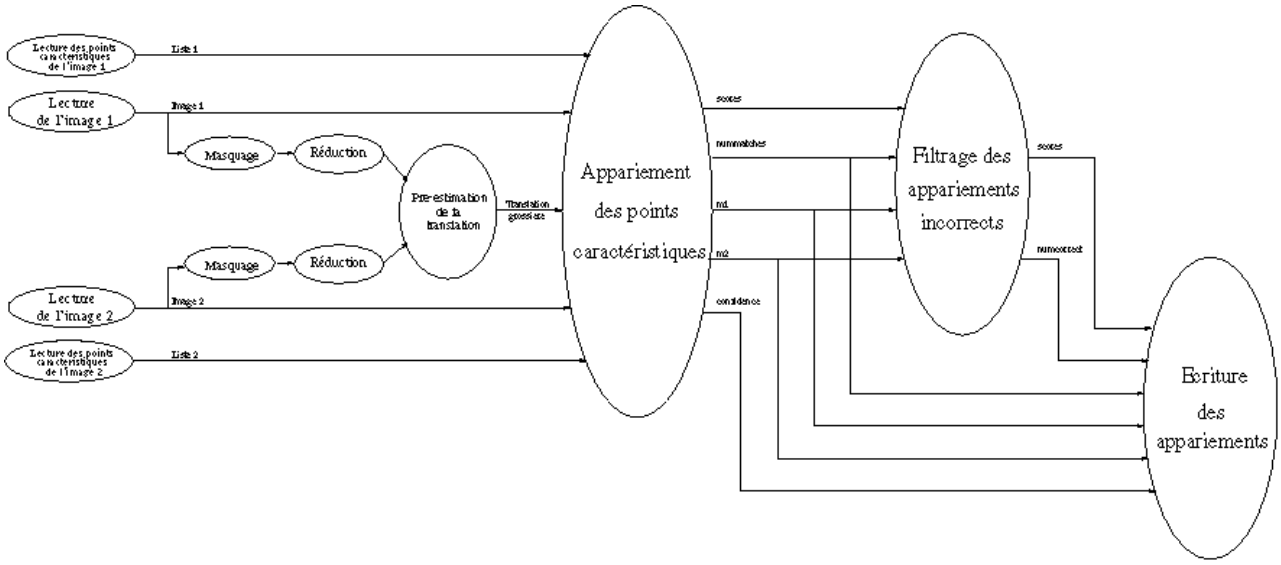


Figure 7 : Spécification simplifiée pour SynDEx v5.

4 Implantation avec SynDEx v5

4.1 Principes de SynDEx

4.1.1 Adéquation Algorithme Architecture

La méthodologie « Adéquation Algorithme Architecture » (AAA) est basée sur des modèles de graphes, autant pour spécifier l'Algorithme et l'Architecture multicomposant, que pour déduire les implantations possibles en terme de transformations de graphes. L'implantation d'un algorithme sur une architecture, consiste à réaliser, en tenant compte des contraintes, une distribution et un ordonnancement des opérations de l'algorithme sur l'architecture caractérisée. L'algorithme est spécifié par un graphe orienté dont chaque sommet est une opération et dont chaque arc est une dépendance de données entre deux opérations. L'architecture est spécifiée par un graphe non orienté dont chaque sommet est de deux types : un opérateur (processeur) séquentialisant des opérations, un communicateur (moyen de communication) séquentialisant des communications, et dont chaque arc est une connexion entre ces sommets. Deux opérateurs ne peuvent être connectés ensemble, idem pour les communicateurs.

L'adéquation consiste à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée.

Cette approche permet de générer automatiquement les exécutifs distribués temps réel pour les composants processeurs, et pour les composants circuits, supportant ensemble (« co-design ») l'exécution de l'algorithme sur l'architecture.

4.1.2 Distribution et ordonnancement optimisés

La distribution consiste tout d'abord à effectuer une partition du graphe de l'algorithme initial, en autant d'éléments de partition qu'il y a d'opérateurs dans le graphe de l'architecture. Il faut ensuite affecter chaque élément de cette partition, c'est à dire chaque sous-graphe du graphe de l'algorithme initial à un opérateur du graphe de l'architecture. Puis il faut effectuer une partition des dépendances de données du graphe de l'algorithme reliant des opérations appartenant à des éléments de partition différents, en autant d'éléments de partition qu'il y a de routes dans le graphe d'architecture. Une route est un chemin dans le graphe d'architecture qui permet de connecter deux opérateurs non directement connectés. Il faut ensuite affecter chaque élément de partition de dépendances de données à une route. Pour chacune de ces dépendances de données inter-partition, il faut créer autant de nouvelles opérations de communication qu'il y a de moyens de communication sur la route sur laquelle elle est affectée, et affecter ces opérations de communication aux communicateurs correspondants de la route.

L'ordonnancement consiste à rendre total (linéariser) l'ordre partiel formé par le sous-graphe des opérations affectées à un opérateur, car celui-ci est une machine séquentielle dont le rôle est d'exécuter séquentiellement des opérations. Il consiste aussi à rendre total l'ordre partiel formé par le sous-graphe des opérations de communications affectées à un communicateur, car celui-ci est aussi une machine séquentielle dont le rôle est d'exécuter séquentiellement des opérations de communication.

Nous allons rechercher parmi toutes les implantations (distribution et ordonnancement) que nous pourrions faire d'un algorithme donné sur une architecture donnée, une implantation particulière que nous considérerons comme optimisée en fonction d'un objectif lié tout d'abord aux contraintes temps réel. Nous pourrons ensuite chercher à minimiser les ressources de cette architecture.

4.1.3 Génération automatique de code

Dès qu'une distribution et un ordonnancement ont été déterminés, il est assez simple de générer automatiquement des exécutifs temps réel, principalement statiques avec une partie dynamique réduite à la prise en compte des données de conditionnement, indiquant si nous devons exécuter ou non une opération ou un sous-graphe d'opérations. Ces exécutifs sont générés en installant un système de communication inter-processeur sans interblocage et ils introduisent un surcoût très faible. La structure du code généré pour chaque processeur est directement issue de la distribution et de l'ordonnancement des calculs de l'algorithme et des communications inter-processeur qui en découlent.

Les exécutifs sont générés dans des fichiers source, un pour chaque mémoire programme de chaque processeur. Chaque fichier est un code intermédiaire composé d'une liste d'appels de macros qui seront traduites par le macro-processeur M4 dans le langage source préféré pour chaque processeur. Le macro-code généré est donc indépendant de l'architecture. Les macros peuvent être classées en deux ensembles. Le premier ensemble est un jeu extensible de macro-instructions spécifiques à développer pour chaque application correspondant aux opérations de l'algorithme. Le second ensemble, appelé noyau générique d'exécutif, est un jeu fixe de macros système qui supportent le chargement initial des mémoires programmes, la gestion mémoire (allocation statique, copies et fenêtres glissantes de macro-registres), le séquencement (sauts conditionnels et itération), les transferts de données inter-processeur (macro-opérations de communication transférant le contenu de macro-registres), les synchronisations inter-séquences (assurant l'alternance entre écriture et lectures de chaque macro-registre partagé entre séquence de calcul et séquences de communication), et le chronométrage (pour permettre la mesure des caractéristiques des opérations de l'algorithme et des performances de l'implantation).

4.2 Implantation simplifiée en monoprocesseur

Nous effectuons dans un premier temps l'implantation simplifiée au niveau de la parallélisation de tâches (pas de parallélisme de données) de notre algorithme avec SynDEx v5 sur une architecture monoprocesseur. Nous avons délibérément fait ce choix afin de débuter l'implantation avec un minimum de difficultés, c'est à dire sans communications inter-processeur. De cette façon, nous pourrons nous assurer de l'équivalence entre la spécification contenant du parallélisme potentiel et la spécification séquentielle et de son bon fonctionnement, en comparant les résultats obtenus à partir du code généré par SynDEx à ceux obtenus avec le programme C de la livraison. Une fois que le fonctionnement de l'algorithme aura été validé de cette manière, nous pourrons déterminer la durée d'exécution de chacune des opérations en générant le code avec l'option « chronométrage ». Ces durées seront utilisées pour caractériser l'algorithme vis à vis de l'architecture afin de visualiser avec SynDEx la distribution et l'ordonnancement donnant une prédiction du comportement en temps réel.

4.2.1 Implantation simplifiée d'une première partie de l'algorithme

Pour implanter notre algorithme, nous procédons de la façon suivante : nous commençons par implanter une première partie de l'algorithme pour valider le bon fonctionnement des toutes premières fonctions. Ainsi, nous pouvons nous appuyer sur ces dernières pour implanter petit à petit l'algorithme complet.

Nous avons choisi d'implanter la partie de l'algorithme correspondant à la lecture de l'image, son masquage, sa réduction puis l'écriture du résultat de la réduction dans un fichier.

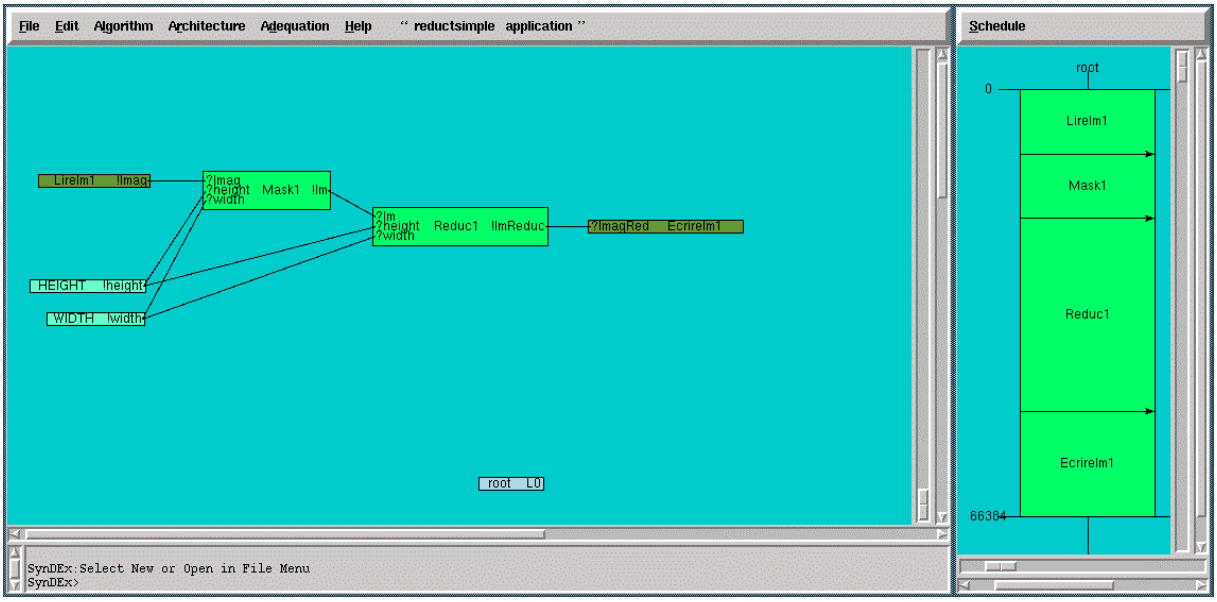


Figure 8 : Implantation d'une première partie de l'algorithme.

La figure 8 représente une copie d'écran de cette application spécifiée avec le logiciel SynDEx v5.

Architecture

L'architecture monoprocesseur correspond à la partie en bas à gauche de la figure. Cette architecture est composée d'un seul sommet (architecture monoprocesseur) appelé « root » (il doit toujours y avoir un sommet `root` dans une architecture) de type U, car nous allons générer du code pour station de travail Unix (la bibliothèque s'appelle U.m4x).

Algorithme

L'algorithme (partie en haut et à gauche) est composé de 6 sommets :

- un sommet d'entrée `LireIm1` de type LireImage1 (qui lit l'image dans un fichier),
- un sommet constante `HEIGHT` de type HEIGHT (qui correspond à la hauteur de l'image en pixels),
- un sommet constante `WIDTH` de type WIDTH (qui correspond à la largeur de l'image en pixels),
- un sommet fonction `Mask1` de type Masquage1 (qui réalise un masquage de l'image),
- un sommet fonction `Reduc1` de type Reduction1 (qui effectue la réduction de l'image),
- un sommet de sortie `EcrireIm1` de type EcrireImage1 (qui écrit l'image réduite dans un fichier).

Remarque :

Chacun des ports d'E/S de chaque sommet doit être connecté à un autre port de même type, et en aucun cas laissé non connecté.

`LireIm1` produit une image (sous forme d'un tableau de caractères non signés « uchar » dont la taille est fonction de `HEIGHT` et `WIDTH`) qui sera ensuite masquée par l'opération `Mask1`. L'image issue de `Mask1` est ensuite réduite d'un facteur 2 (la taille du tableau sera donc divisée par 4 car la hauteur `HEIGHT` et la largeur `WIDTH` de l'image seront divisée chacune par deux). L'image ainsi réduite (tableau d'entiers courts non signés « ushort » dont la taille est fonction de `HEIGHT`/2 et `WIDTH`/2) est enregistrée par `EcrireIm1` dans un fichier.

Caractérisation

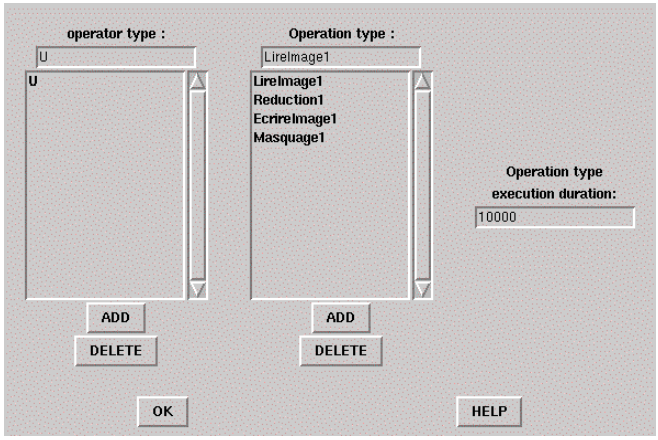


Figure 9 : Caractérisation de l'application.

Pour que SynDEx puisse faire une distribution/ordonnancement (ici simplement un ordonnancement car nous sommes en monoprocesseur), il ne suffit pas de créer les opérations et de les connecter, il faut caractériser l'application (Fig. 9). C'est à dire, attribuer à chacun des types d'opérations un temps d'exécution que nous aurons estimé, mais nous verrons au §4.2.5 que ces valeurs peuvent aussi être issues de mesures. L'unité utilisée pour spécifier les durées est libre. Les sommets constantes ne seront pas caractérisés car ils sont exécutés une seule fois lors de l'initialisation. En effet, pour toute ré-exécution de l'algorithme, ils produisent la même valeur, leur durée est donc considérée comme nulle. Ces sommets n'apparaîtront pas lors de la distribution/ordonnancement.

Exemple :

Sur la figure 9, nous pouvons voir que la durée d'exécution du sommet de type LireImage1 est de 10000 unités de temps si ce sommet est exécuté par un processeur de type U.

Adéquation

Une fois la description de l'algorithme et de l'architecture, et la caractérisation de l'algorithme effectuées, nous pouvons lancer l'adéquation. Le résultat de l'adéquation (Fig. 8 à droite, « Schedule ») représente la distribution/ordonnancement (simplement ordonnancement puisque l'architecture est monoprocesseur) des opérations de l'algorithme sur chaque processeur de l'architecture c'est à dire pour chaque processeur une séquence d'opérations. Toutes les opérations y sont représentées par des rectangles dont la hauteur est proportionnelle à leur temps d'exécution. SynDEx a calculé que le temps d'exécution total est de 66384 unités de temps. Cela est indiqué au bas de l'ordonnancement.

Génération de macro-code

Il est très simple, une fois l'ordonnancement effectué, de générer automatiquement le macro-code de cette application. En effet, la génération des exécutifs est obtenue en cliquant dans le menu « Adéquation » sur « Genexec ». Lors de la génération du macro-code deux fichiers dont les noms sont suffixés par « .m4 » apparaissent : « nom de l'appli. ».m4 (contenant la topologie de l'architecture, il sera utilisé pour générer un makefile automatisant la compilation) et `root`.m4 (contenant le macro-code correspondant au processeur `root`). Ces deux fichiers seront utilisé dans la chaîne de compilation (Cf. §4.2.2).

4.2.2 Chaîne de compilation

La chaîne de compilation (Fig. 10) est l'ensemble des transformations qui permettent de générer et d'exécuter le code de l'application à partir du macro-code automatiquement généré par SynDEx.

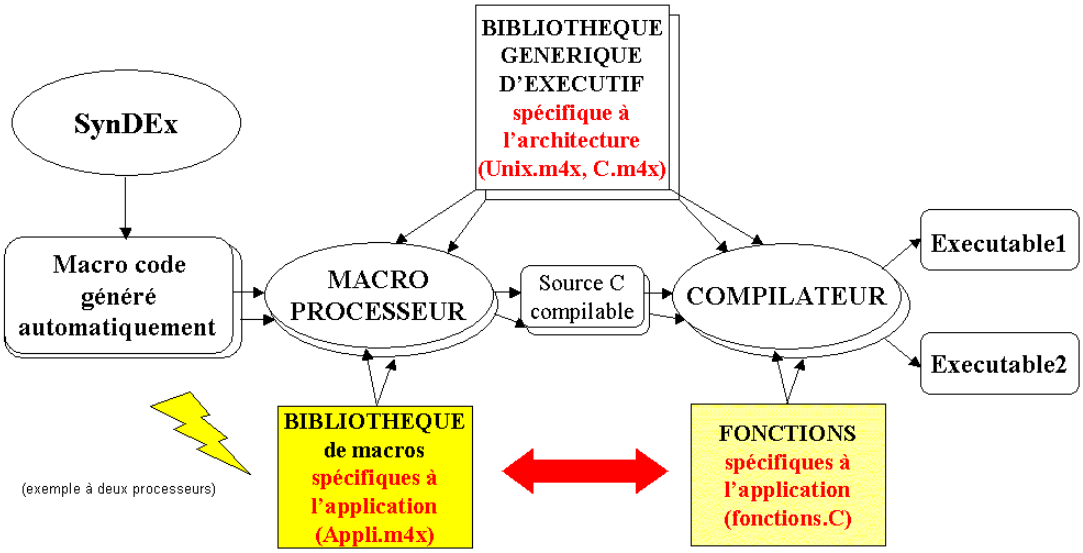


Figure 10 : Chaîne de compilation.

Le macro processeur (ici nous utilisons GNU M4), explore le macro-code généré par SynDEx (comme nous l'avons vu, un fichier de macro-code pour chaque processeur), et remplace les différentes macros qu'il rencontre par leurs définitions qu'il trouve dans les bibliothèques génériques d'exécutifs (spécifiques à l'architecture) et les bibliothèques spécifiques à l'application. Les bibliothèques génériques d'exécutifs que nous utilisons sont U.m4x (pour les stations de travail sous UNIX) et C.m4x (le langage utilisé est le C). La bibliothèque spécifique à notre application est « nom de l'appli ».m4x (au §4.2.3 nous verrons comment l'écrire). A l'issue du travail effectué par le macro processeur, nous disposons d'un source C compilable pour chaque processeur. La suite de la compilation est faite « classiquement » à l'aide d'un compilateur correspondant au processeur et à l'aide de ses bibliothèques. Il génère un binaire exécutable.

4.2.3 Association opération-SynDEx/fonction-C

Étant donné que notre application utilise des fonctions spécifiques écrites en C, ne faisant pas partie des bibliothèques génériques d'exécutifs, il faut écrire une bibliothèque de macros spécifiques à l'application. Ces macros vont remplacer le macro-code par des appels à des fonctions C spécifiques elles aussi à l'application et compilées séparément.

Pour chaque type d'opération SynDEx, nous écrivons une macro qui va être substituée par l'appel à la fonction C appropriée. Bien sûr ceci nécessite un découpage du programme C fourni, et la création du fichier « nom de l'appli ».m4x.

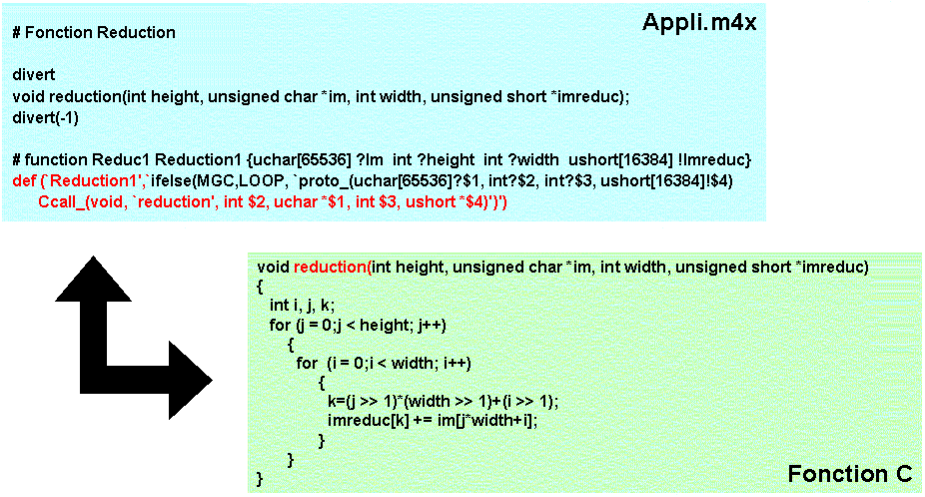


Figure 11 : Association d'une opération SynDEx et d'une fonction C.

Par exemple, sur la figure 11, nous associons le type d'opération « Reduction1 » à la fonction C « reduction » spécifique à l'application (écrite dans un fichier « .c », partie basse Fig. 11) par la définition « `def` » (extrait, partie haute de la Fig. 11) que nous placerons dans la bibliothèque spécifique à l'application (fichier « nom de l'appli ».m4x). Nous pouvons voir que dès que la macro « Reduction1 » est rencontrée, nous faisons appel à la macro `Ccall` (définit dans le noyau générique C.m4x) qui permet de générer l'appel d'une fonction C. Dans les arguments de la macro `Ccall`, nous déterminons le nom de la fonction C à appeler (ici « reduction ») et dans l'ordre les arguments de la fonction C appelées. Il ne faut pas oublier d'écrire le prototype de la fonction appelée entre « divert » et « divert(-1) », cette ligne apparaîtra au début du fichier source C compilable.

4.2.4 Tests et validation incrémentale

Nous allons effectuer les différents tests et la validation de notre application (calcul de translation entre deux images) selon la méthode décrite ci-après.

Méthode

Dans la section 2.3, nous avons testé l'application en séquentiel afin de constituer des fichiers de sorties de référence. Pour valider la bonne implantation de la partie d'algorithme que nous avons choisi, nous comparons tout simplement les fichiers de sorties de référence aux fichiers obtenus avec SynDEx.

- Si les fichiers sont différents, il faut procéder à une vérification (sur « nom de l'appli.m4x » et les bibliothèques de fonctions C) suivie d'une correction pour procéder à nouveau à la comparaison des fichiers. Tant que les fichiers sont différents nous recommençons ce processus.
- Si les fichiers sont identiques, l'implantation de la partie d'algorithme est validée et nous pouvons passer à la suite.

Comme nous l'avons vu (Cf. §4.2.1), il s'agit d'une validation incrémentale. Donc, nous allons rajouter une nouvelle opération SynDEx, compléter la caractérisation (dans « nom de l'appli ».sdx) et la bibliothèque de notre application (« nom de l'appli ».m4x), puis valider comme précédemment jusqu'à l'implantation complète de l'algorithme.

Tests et validation pour notre application

Au §4.2.1, nous avons implanté une première partie de l'algorithme (Fig. 8). Nous procédons à présent à l'implantation complète de notre algorithme selon la méthode que nous venons de décrire.

Ainsi, nous implantons l'algorithme de réduction des deux images (Fig. 12), qui lit dans un premier temps les deux images, les masque, les réduit d'un facteur deux. Il termine par l'enregistrement des deux images ainsi réduites dans deux fichiers. Nous complétons la caractérisation et la bibliothèque spécifique à notre application avec les nouveaux appels de fonctions.

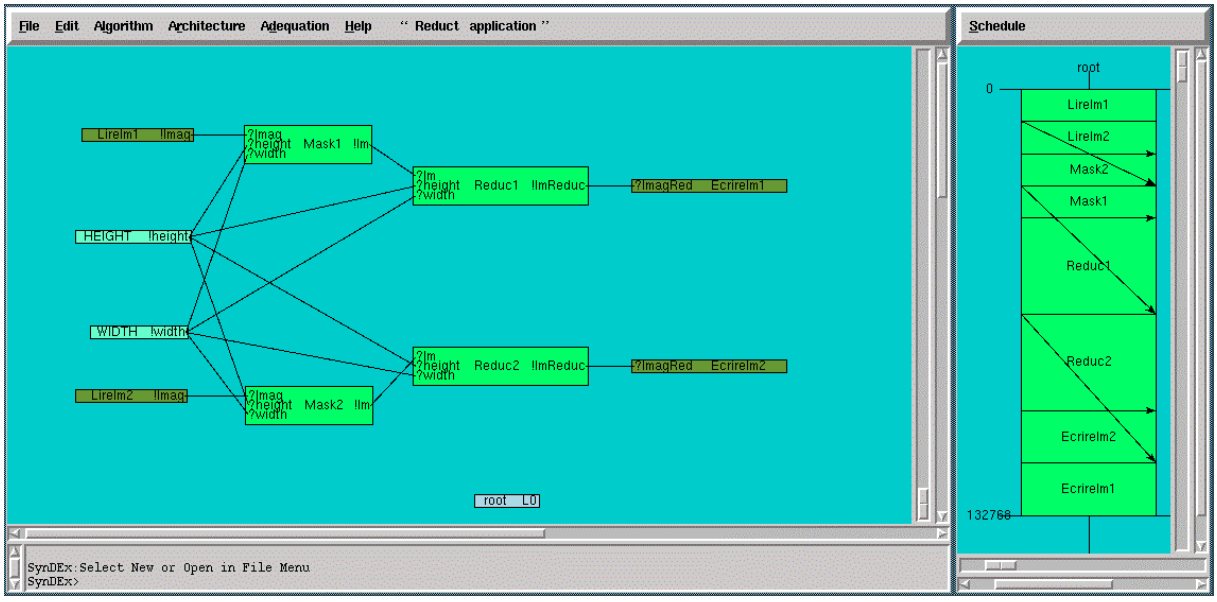


Figure 12 : Implantation de l'algorithme jusqu'à la réduction des deux images.

Ensuite, nous implantons l'opération de pré-estimation (Fig. 13) qui à partir des deux images réduites détermine une translation estimée (translation grossière). Nous complétons la caractérisation et la bibliothèque spécifique à notre application avec les nouveaux appels de fonctions.

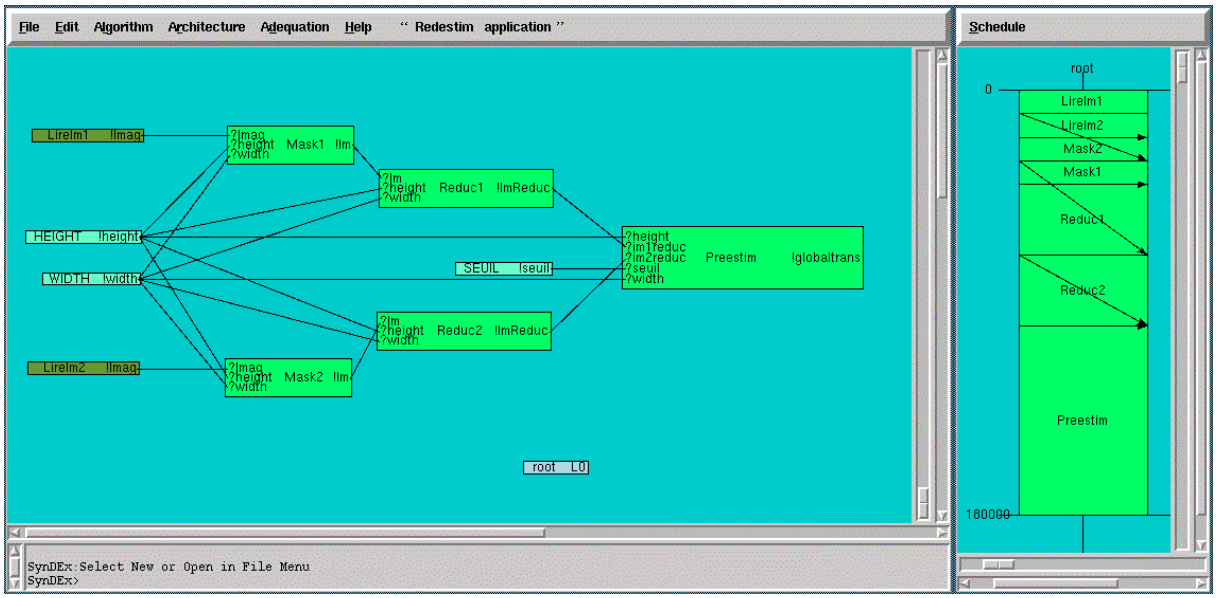


Figure 13 : Implantation de l'algorithme jusqu'à la pré-estimation de la translation.

Puis, nous implantons les opérations `Plots1`, `Plots2` (pour la lecture des fichiers contenant les points caractéristiques extraits des 2 images) et `Appar` qui permettent d'effectuer l'appariement des points caractéristiques en fonction des paramètres `CONF`, `DISTX`, `DISTY`, `VAR`, `WIN` et la translation grossière (Fig. 14). Nous complétons la caractérisation et la bibliothèque spécifique à notre application avec les nouveaux appels de fonctions.

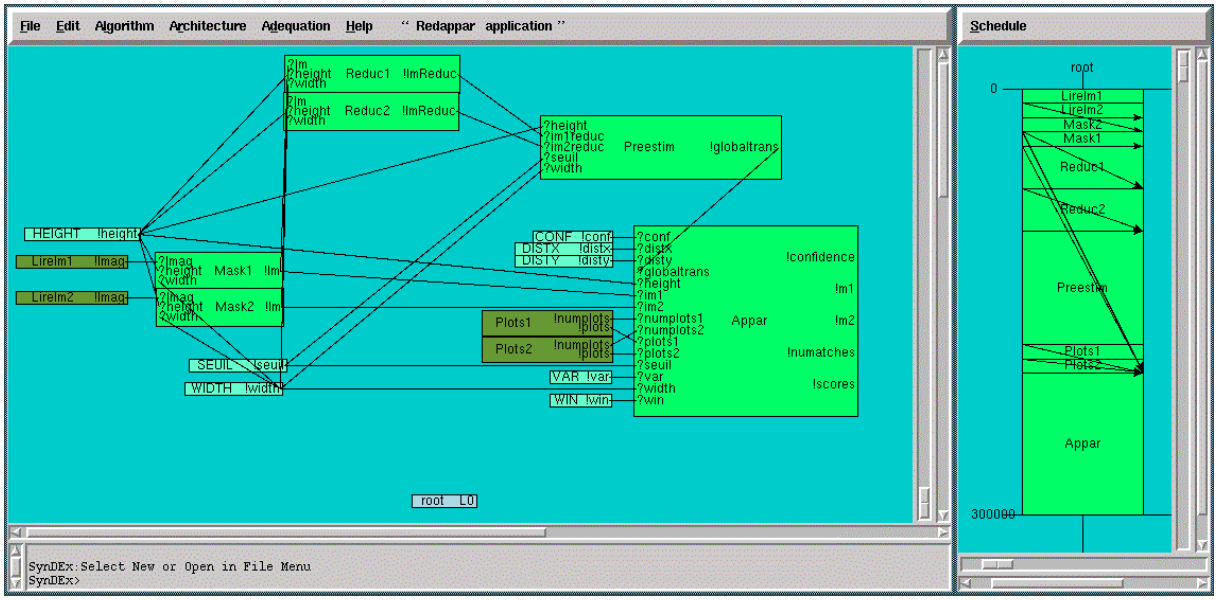


Figure 14 : Implantation de l'algorithme jusqu'à l'appariement des points caractéristiques.

Enfin, nous implantons l'opération de filtrage des appariements incorrects (`FiltrApparInc`) et celle d'écriture des appariements (Fig. 15) dans un fichier après filtrage. Nous complétons la caractérisation et la bibliothèque spécifique à notre application avec les nouveaux appels de fonctions (Annexe 2).

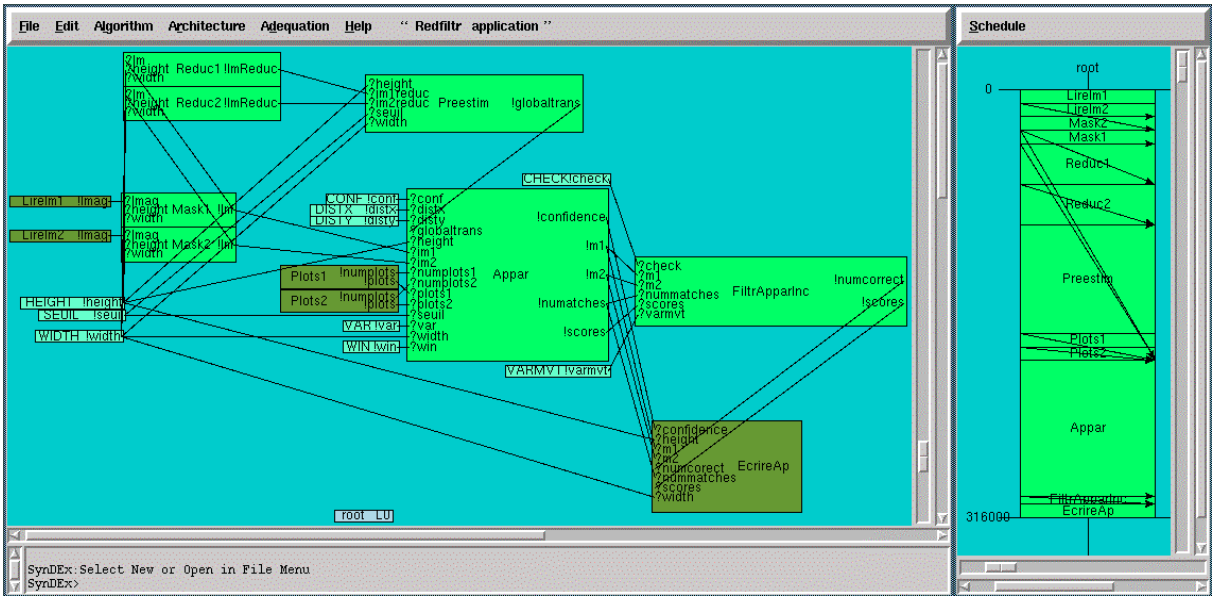


Figure 15 : Implantation de l'algorithme jusqu'au filtrage des appariements incorrects.

4.2.5 Caractérisation par la mesure

Dans les étapes précédentes, nous avons implanté l'algorithme et validé son fonctionnement. À présent, nous allons effectuer le chronométrage des différentes opérations afin de caractériser au mieux les durées. Ces dernières seront très importantes en multi-processeurs (Cf. §4.3), car elles seront utilisées par l'heuristique de distribution/ordonnancement de SynDEx afin de minimiser la durée d'exécution de l'application.

Pour ajouter le chronométrage au code, il suffit de se rendre dans le menu « Adequation », choisir l'option « with Chrono » et demander la génération de macro-code avec « GenExec » (dans le même menu). Des opérations de chronométrage sont alors insérées automatiquement dans le code généré pour mesurer la date de début et de fin de chaque opération.

Ainsi, après l'exécution du programme, les résultats des mesures sont affichées automatiquement sous la forme de quelques lignes de texte dans la fenêtre du terminal (Fig. 16). Ces dernières représentent des labels (un par opération), chacun suivi d'une date (date de fin) et de la durée d'exécution (date de fin - date de début).

```
Chronometric profiling:
label date dt=date-datePrec-chronoOverhead (microseconds)
1001 453705274 chronoOverhead=1
1002 453705802 527
1003 453708275 2472
1004 453710722 2446
1005 453715995 5272
1006 453720405 4409
1007 454201925 481519
1008 454204444 2518
1009 454206064 1619
1010 454864253 658188
1011 454867663 3409
1012 454927254 59590
Use `grep "^chrono" *.c' to collect labels number/name correspondences
```

Figure 16 : Chronométrage des opérations.

Pour obtenir la correspondance entre les temps d'exécution et les opérations (Fig. 17), il suffit de taper cette commande : `grep``chrono'' *c` qui retourne la correspondance entre le label et le nom de l'opération.

```
root.c:chrono 1001 = root_LireIm1
root.c:chrono 1002 = root_LireIm2
root.c:chrono 1003 = root_Mask2
root.c:chrono 1004 = root_Mask1
root.c:chrono 1005 = root_Reduc1
root.c:chrono 1006 = root_Reduc2
root.c:chrono 1007 = root_Preestim
root.c:chrono 1008 = root_Plots1
root.c:chrono 1009 = root_Plots2
root.c:chrono 1010 = root_Appar
root.c:chrono 1011 = root_FiltrApparInc
root.c:chrono 1012 = root_EcrireAp
```

Figure 17 : Correspondances entre temps et opérations.

Exemple :

La 5ème ligne de la figure 16, débute par le label de l'opération (1003), qui est suivi de la date de fin de cette opération, et se termine par le temps d'exécution de l'opération (2472). Pour connaître le nom de l'opération, nous exécutons la commande `grep``chrono'' *c` pour obtenir les correspondances. Nous pouvons constater sur la figure 17 qu'il s'agit de l'opération Mask2 qui est exécutée sur le processeur `root`.

Nous connaissons à présent les temps d'exécution des différentes opérations, nous ré-injectons ces temps dans SynDEx pour obtenir l'implantation de notre algorithme de calcul de translation entre deux images avec les temps d'exécution (Fig. 18).

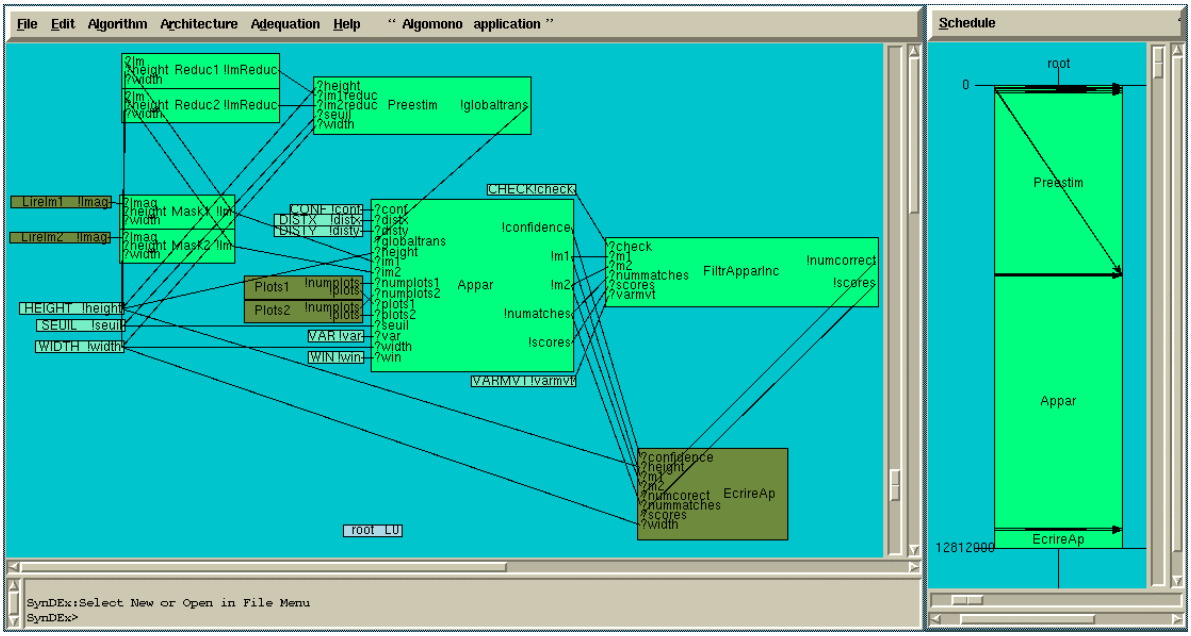


Figure 18 : Algorithme en monoprocesseur.

4.3 Implantation simplifiée de l'algorithme en multi-stations de travail

Après avoir vérifié le bon fonctionnement de l'implantation de notre algorithme en monoprocesseur (Cf. §4.2.4), nous pouvons passer à une implantation multi-processeurs. Pour cela, nous allons reprendre l'implantation en monoprocesseur de notre algorithme (Fig. 18) à laquelle nous ajouterons, dans le graphe d'architecture, un nouveau processeur (du même type pour le moment) que nous relierons au premier par un média de communication.

Le passage effectif de l'implantation de l'algorithme en monoprocesseur à l'implantation en multi-processeurs (Fig. 19) est très simple, puisqu'il s'agit d'ajouter un processeur (menu « Architecture », « New operator ») de même type (les durées d'exécution seront les mêmes sur les deux processeurs, c'est à dire la même caractérisation) avec une E/S de type TCP (comme le 1er processeur) et un média de communication (menu « Architecture », « New bus ») de type TCP (la bibliothèque TCP.m4x sera utilisée lors de la compilation) qu'il faudra caractériser (durée de communications pour chaque type de données).

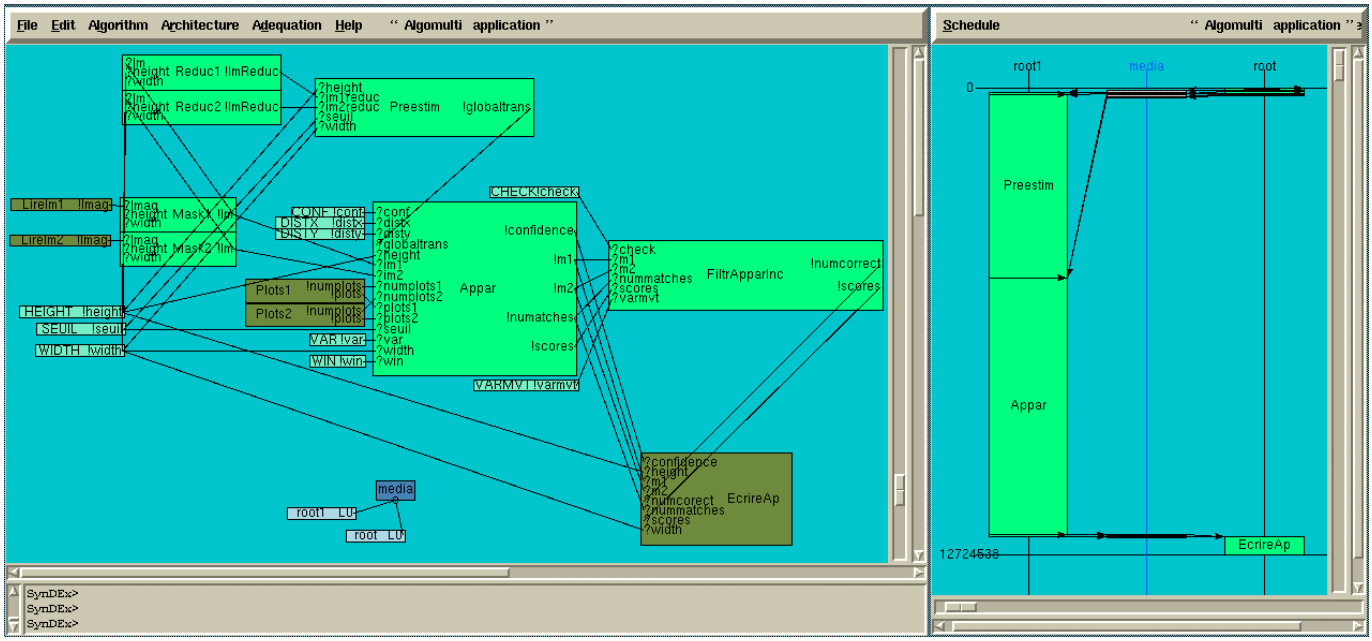


Figure 19 : Algorithme en multiprocesseur.

En monoprocesseur, nous avons mesuré les temps d'exécution de chaque opération. Il est important pour l'implantation de notre algorithme en multi-processeurs d'utiliser ces valeurs, mais aussi de spécifier la valeurs des durées de communications pour chaque type de données utilisé. Toutes ces durées (opérations de calcul, communications) seront utilisés par l'heuristique de SynDEx pour effectuer la distribution/ordonnancement afin de minimiser la durée d'exécution de notre application.

Exemple :

La partie droite de la figure 19 (agrandie en Fig. 20) présente le résultat de la distribution ordonnancement de l'algorithme (partie haute gauche) sur l'architecture bi-processeur (partie basse gauche) sous la forme d'un diagramme temporel. Comme expliqué précédemment (au §4.2.1 Adéquation), les opérations de calculs exécutées par chaque processeur sont représentées par des rectangles, de même que les opérations de communication qui ont été insérées automatiquement par SynDEx pour transférer les données quand cela était nécessaire (Cf. colonne « média » de la Fig. 20). Sur la figure 20, nous constatons par exemple que

`Preestim` et `Appar` sont exécutées par `root1`, et que `Preestim` utilise des données produites par l'opération `Reduc1` exécutée par `root`.

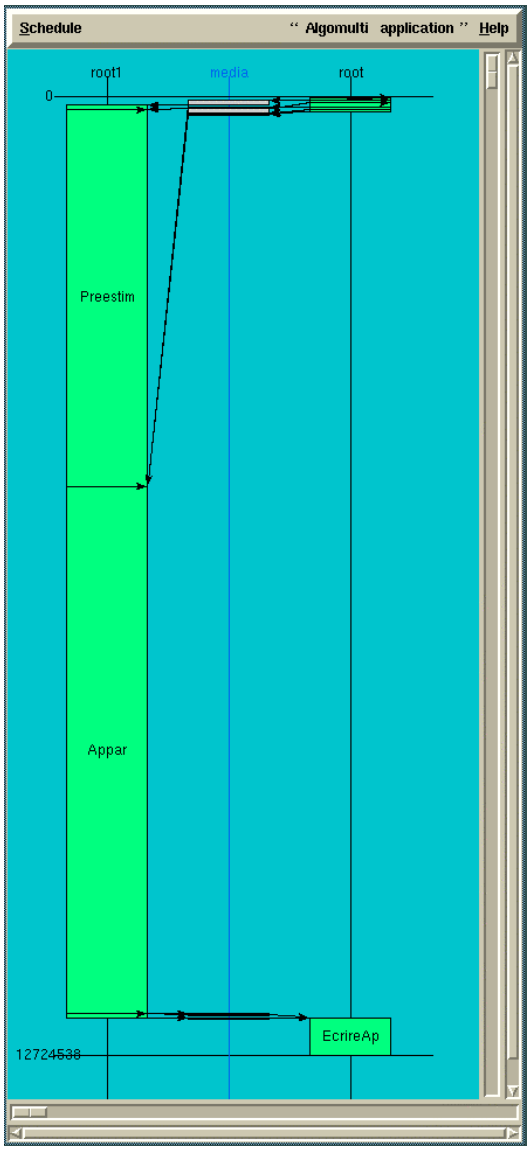


Figure 20 : Distribution/ordonnancement de l'algorithme en bi-processeur.

Les durées des communications sont fondamentales pour une implantation multi-processeurs. Par exemple, si nous faisons varier les durées des communications, l'heuristique de SynDEx peut fournir des distribution/ordonnancement très différents. Sur l'exemple d'application présenté sur la figure 19, nous constatons que le coût des communications est très faible (quelques millisecondes) vis à vis des durées de calcul. Sur l'exemple d'application de la figure 21, nous avons lancé l'heuristique en utilisant les même processeurs que précédemment (Fig. 19), mais en utilisant un média de communication dix fois plus lent. Le coût des communications étant alors très élevé vis à vis des calculs, l'heuristique a distribué et ordonnancé toutes les opérations sur le même processeur `root`. Grâce à SynDEx, il est donc possible d'implanter rapidement un algorithme sur différentes architectures, pour rechercher la plus adaptée. Nous pouvons prédire les performances de l'implantation de l'algorithme sur différentes architectures même si nous ne les possédons pas physiquement.

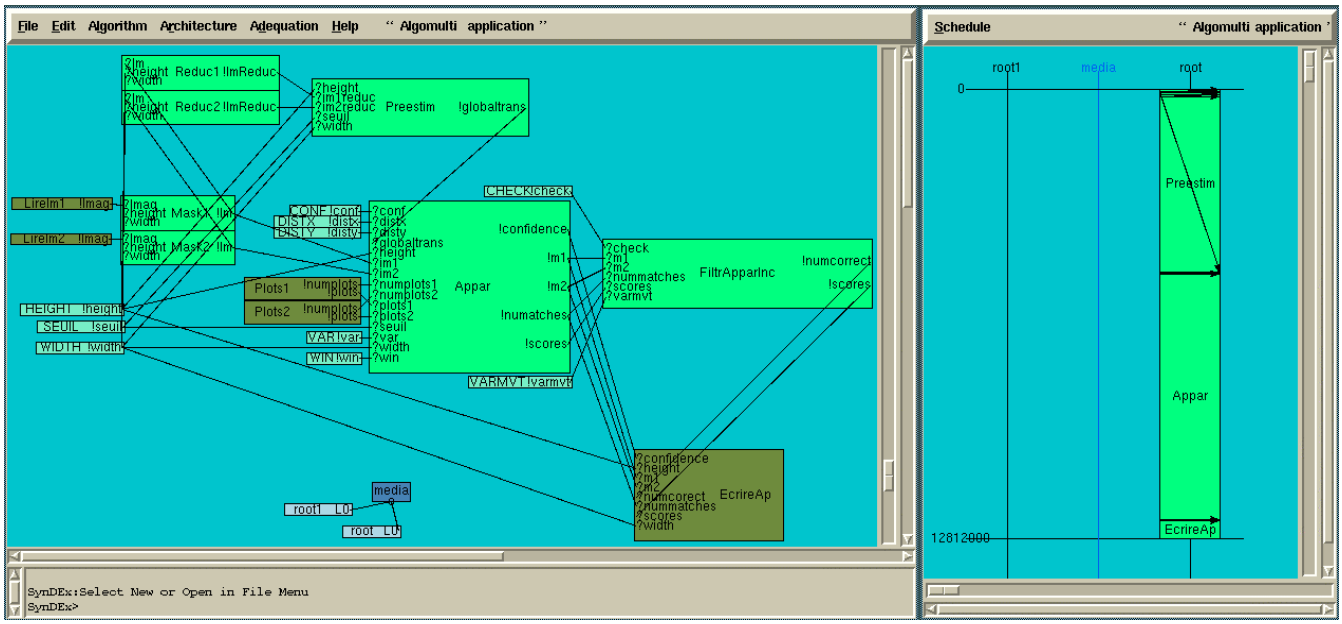


Figure 21 : Influence des durées de communications sur la distribution/ordonnancement.

4.4 Conclusion

Nous obtenons pour l'implantation en mono-processeur de notre algorithme, de calcul d'une translation entre deux images, une durée d'exécution de 12.812.000 unités de temps (Fig. 18) et une durée d'exécution de 12.724.538 unités de temps en bi-processeur (Fig. 20). Nous avons donc réalisé un gain de 0,68%, ce qui est extrêmement faible.

Pour espérer réduire significativement la durée d'exécution de notre algorithme sur une architecture multi-processeurs, il est nécessaire d'exprimer plus de parallélisme potentiel. Quelque soit le type d'architecture utilisé (mono-station ou multi-stations de travail), la durée d'exécution des opérations `Preestim` et `Appar` est prépondérante : plus de 90 % de la durée totale d'exécution de l'algorithme complet. Il faut donc raffiner la spécification de ces deux opérations afin de faire apparaître du parallélisme de données et d'opérations supplémentaires. Grâce à la nouvelle version (SynDEx v6) du logiciel nous allons pouvoir spécifier ces deux opérations en utilisant : la « description hiérarchique » qui n'existait pas dans la version 5 de SynDEx, ainsi que la « factorisation » et le « conditionnement » (Cf. prochain chapitre) afin de mettre en évidence plus de parallélisme potentiel et exploiter au mieux les architectures multi-processeurs.

5 Implantation avec SynDEx v6

5.1 SynDEx v6

Le passage de la version 5 à la version 6 du logiciel SynDEx, apporte de nouvelles fonctionnalités permettant de spécifier le graphe de l'algorithme de manière hiérarchique, de conditionner l'exécution de certaines opérations et de répéter l'exécution d'une même opération plusieurs fois sur des données différentes. En conséquence, les heuristiques d'optimisation de la distribution/ordonnancement de la nouvelle version du logiciel (SynDEx v6) exploitent ces nouvelles fonctionnalités afin de construire une implantation qui minimise la durée d'exécution de l'algorithme.

La « **spécification hiérarchique** » permet de représenter l'algorithme complet de façon plus claire. Un ensemble d'opérations qui réalisent une fonctionnalité complexe peut être encapsulées dans une seule opération. Nous avons la possibilité de voir à tout moment la composition de cette opération, en cliquant tout simplement dessus. Graphiquement, nous pouvons repérer rapidement les opérations hiérarchiques car elles sont représentées par une boîte superposée à une autre (formant ainsi une boîte avec une ombre).

La « **factorisation** » permet de mettre en évidence du parallélisme de données (même opération répétée sur des données différentes). Le nombre de répétitions d'une opération est indiqué entre crochets à côté du nom de cette même opération. Le nombre de répétitions peut être modifié très rapidement en accédant au nom de l'opération, soit par une valeur littérale, soit par un paramètre variable.

L'heuristique est capable de prendre en compte les cas où les opérations sont exécutées dans des cas exclusifs (jamais en même temps) afin de minimiser la durée d'exécution. Il est facile de repérer graphiquement une opération conditionnée. En effet, le nom de son entrée de « **conditionnement** » est surligné en gris.

D'autres modifications ont été apportées :

- Le graphe d'architecture n'est plus représenté sur la même fenêtre que le graphe de l'algorithme. Ce qui permet d'ajouter autant de graphes d'architecture que l'on désire à un même fichier, avec la possibilité de choisir l'architecture que l'on désire à tout moment.
- Des arguments sont à présent disponibles afin de paramétrer des constantes, des tailles de ports et des nombres de répétitions d'opérations.
- La caractérisation des durées des différentes opérations est plus simple à présent, car elle est représentée sous la forme d'une liste avec la possibilité de changer plus facilement les temps.
- Remarquons enfin que, dans la représentation temporelle de l'implantation, le temps correspond maintenant à l'axe horizontal.

Remarque :

La génération de macro code et les moyens de communications par mémoire partagée étant en cours de développement et non disponible dans la version 6 de SynDEx à ma disposition, je n'ai pas pu générer le macro-code et simuler l'algorithme sur une architecture à mémoire partagée.

5.2 Implantation de l'algorithme

5.2.1 Implantation de l'algorithme sans parallélisme de données

Pour prendre en main et s'assurer du bon fonctionnement de la nouvelle version du logiciel SynDEx, nous avons effectué le portage de l'implantation de l'algorithme complet spécifié en v5 sur la v6 (Fig. 22 et Fig. 23). C'est à dire que nous avons décrit le même algorithme, la même architecture (bi-processeur) et nous avons caractérisé de façon identique les opérations de l'algorithme.

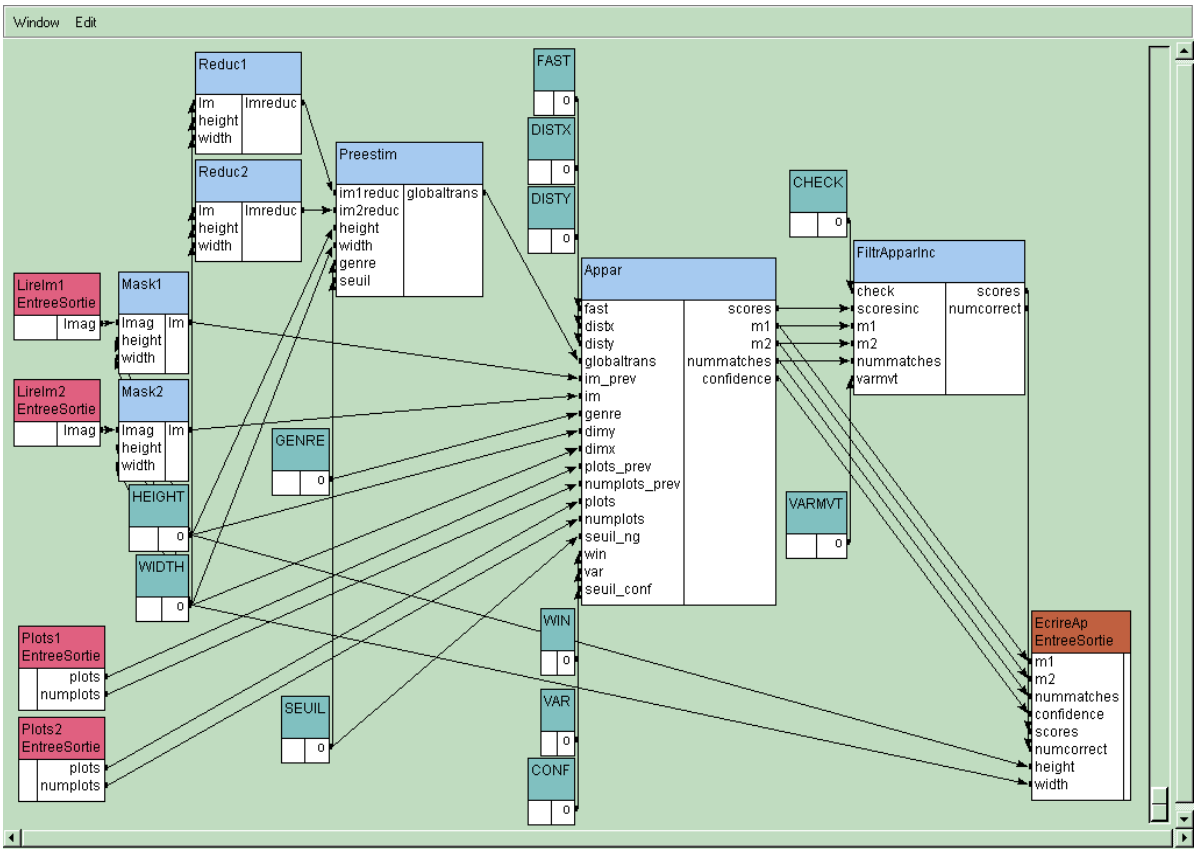


Figure 22 : Algorithme de calcul de translation entre deux images.

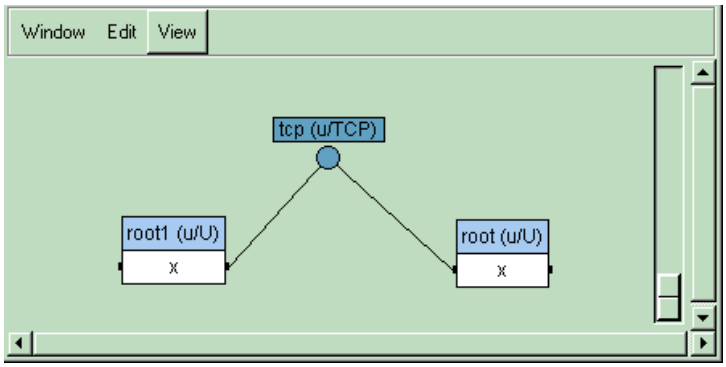


Figure 23 : Architecture bi-processeur.

Après l'adéquation (item « No Flatten » dans le menu «Adequation») de l'algorithme sur l'architecture, nous obtenons la même distribution/ordonnancement (Fig. 24) qu'avec la v5 du logiciel (Fig. 20). Les temps d'exécution de notre algorithme sur les deux versions du logiciel sont les mêmes (Cf. §4.4).

Figure 24 : Distribution/ordonnancement de l'algorithme en bi-processeur.

Une fois cette étape indispensable validée, notre objectif est de réduire la durée d'exécution totale de notre algorithme sur une architecture multi-processeurs. Pour cela, il faut raffiner la spécification des deux opérations (`Preestim` et `Appar`) afin de faire apparaître du parallélisme potentiel de données et du parallélisme potentiel d'opérations supplémentaires.

Nous débutons par la spécification hiérarchique de l'opération `Preestim`, puis nous terminerons par l'opération `Appar`. La spécification de ces opérations ce fait selon la méthode décrite ci-après.

5.2.2 Méthode de spécification hiérarchique des opérations

Pour être certain de la bonne spécification hiérarchique de l'opération, il faut procéder à une implantation de façon progressive par niveaux. Pour spécifier hiérarchiquement une opération, il faut bien analyser l'algorithme programmé en C, afin de découper en quelques fonctions simples, la fonction associée à l'opération. De cette façon, nous maintenons une certaine clarté dans la lecture du graphe de l'algorithme.

Lorsque nous découpons l'algorithme, certaines modifications plus ou moins importantes doivent être apportées afin de faire apparaître plus de parallélisme de données.

Si nous rencontrons dans le programme C une même opération exécutée sur des données différentes (parallélisme potentiel de données) nous utilisons les répétitions.

Si une boucle doit être exécutée, à partir d'une valeur jusqu'à une autre, nous pouvons rendre effectif le parallélisme potentiel de données en générant toutes ces valeurs dans un vecteur (Cf. §5.2.3).

La présence du conditionnement « if(…) continue » dans le corps de la fonction à répéter est gênante, car chaque opération répétée doit absolument renvoyer une valeur (qu'elle soit valide ou non) et être suivie d'une opération de sélection. Tous les conditionnements de ce type doivent donc être placés avant l'opération répétée, (dans le cas où les résultats de l'opération ne seraient pas utilisés par le conditionnement) ou après l'opération répétée (dans le cas où les résultats de l'opération seraient utilisés par le conditionnement), mais les résultats devront alors être accompagnés d'une indication sur leur validité (Cf. §5.2.3).

5.2.3 Spécification hiérarchique de l'opération Preestim

L'algorithme fourni en annexe et programmé en langage C, renferme du parallélisme de données au niveau de la corrélation. Mais avant d'y arriver, il faut spécifier l'opération `Preestim`.

La fonction de pré-estimation de l'algorithme programmé en C peut être aisément découpée, dans un premier temps, en 3 opérations (Fig. 25) :

- `ParametresPreestim`, qui effectue le calcul des paramètres de corrélation pour la pré-estimation à partir des constantes `HEIGHT`, `WIDTH` et `SEUIL`,
- `CorrelationFineP`, qui effectue les calculs de corrélation sur les images réduites en fonction des paramètres qui lui sont fournis. Cette opération permet de trouver les nouvelles coordonnées du centre de l'image réduite 1 sur l'image réduite 2,
- `TranslationGlobale`, qui à partir des coordonnées du centre de l'image réduite 1 et de ces nouvelles coordonnées sur l'image réduite 2, détermine la translation globale estimée entre les deux images.

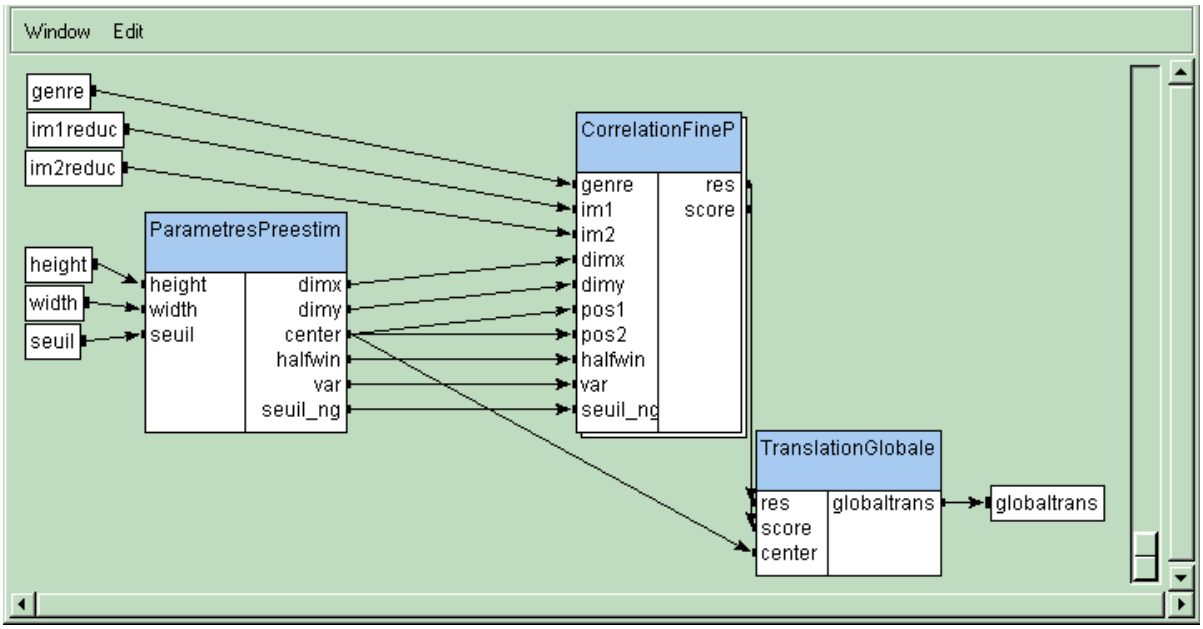


Figure 25 : Spécification hiérarchique de niveau 1 de l'opération Preestim.

Nous pouvons voir sur la figure 25 que l'opération `CorrelationFineP` est ombrée ce qui, comme nous l'avons vu au §5.1, signifie qu'elle est spécifiée hiérarchiquement.

Pour réaliser les calculs de corrélation (Annexe 3), il faut ajouter un nouveau niveau à la spécification hiérarchique de l'opération `Preestim`. Pour cela, nous allons spécifier plus finement l'opération `CorrelationFineP` (Fig. 26). Cette dernière peut se décomposer en deux opérations :

- `CalculDeplacements`, qui effectue le calcul des déplacements autorisés pour les corrélations à partir des coordonnées du centre de l'image réduite 2 et des paramètres `dimy`, `dimx` et `halfwin`,
- `CorrelationsDeplacements`, qui effectue les calculs de corrélations entre les images réduites 1 et 2 selon les déplacements `di_min` , `di_max`, `dj_min` et `dj_max`. Puis, qui recherche les coordonnées du pic de corrélation (`res`) avec le score correspondant (`score`).

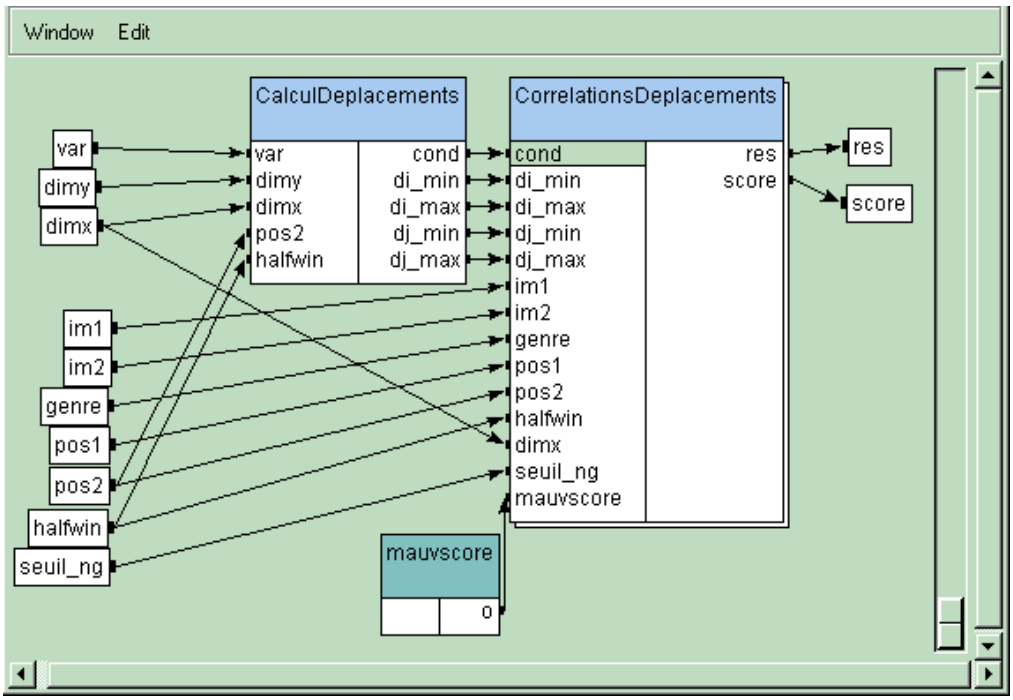


Figure 26 : Spécification hiérarchique de l'opération CorrelationFineP (niveau 2 de Preestim).

Nous pouvons remarquer sur la figure 26 que l'entrée `cond`, surlignée en gris, est une entrée de conditionnement. L'opération `CorrelationsDeplacements` est donc conditionnée par son entrée `cond`. Il existe deux cas au conditionnement de cette opération:

- Un cas `cond` = 0 (Fig. 27), l'opération de corrélation n'est pas réalisée et la sortie `res` n'est pas valide,
- Un autre `cond` = 1 (Fig. 28), l'opération de corrélation est réalisée selon la description précédente et la sortie `res` contient les coordonnées du pic de corrélation.

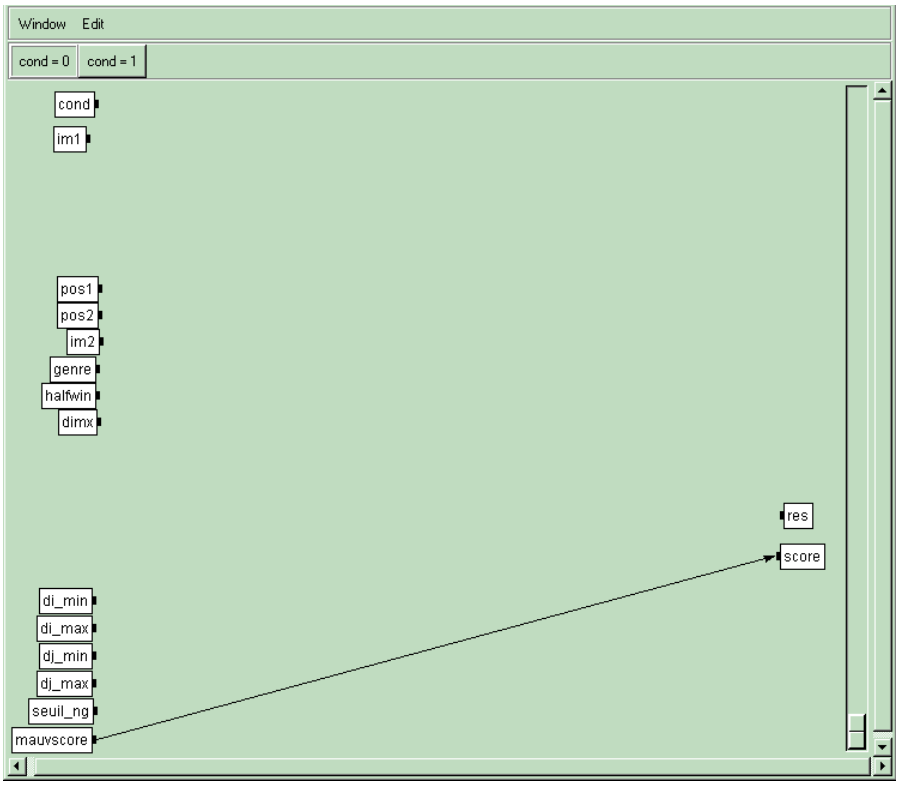


Figure 27 : Spécification de l'opération conditionnée CorrelationsDeplacements (cond = 0).

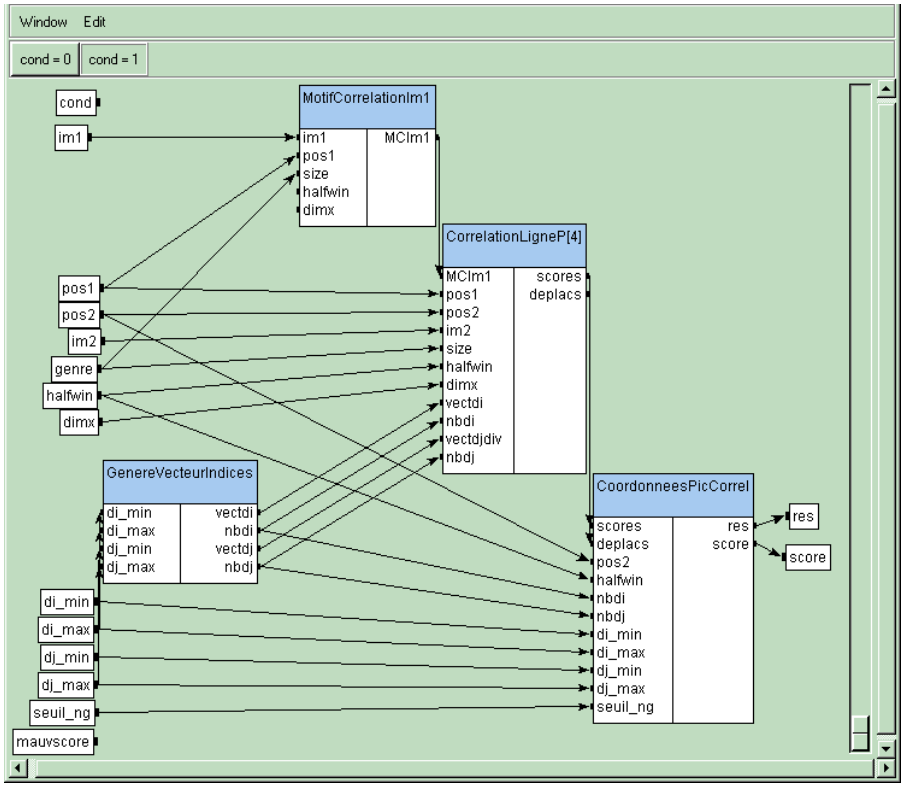


Figure 28 : Spécification de l'opération conditionnée CorrelationsDeplacements (cond = 1).

Nous pouvons voir sur la figure 28, que le nom de l'opération `CorrelationLigneP` est suivi d'un nombre mis entre crochets (ici 4) ce qui signifie qu'elle est répétée 4 fois.

Dans l'algorithme programmé en C, la corrélation se présente sous la forme d'une opération (Fig. 29) qui a pour entrées les images réduites 1 et 2 ainsi que deux couples (min., max.) de déplacements en abscisse (di_min, di_max) et en ordonnée (dj_min, dj_max), et pour sorties les scores de corrélation pour chacun des déplacements.

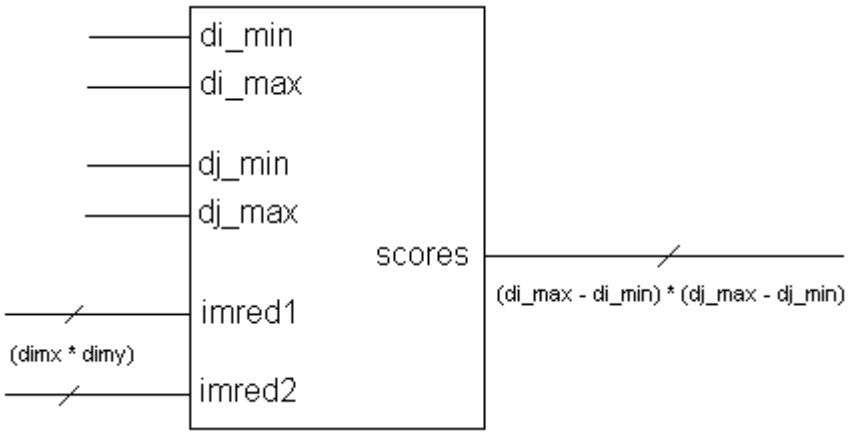


Figure 29 : Opération CorrelationsDeplacements.

La même opération de corrélation est exécutée sur les différents déplacements (parallélisme potentiel de données). C'est pour cela que nous avons choisi d'utiliser les répétitions à cet endroit précis. Pour faire apparaître les répétitions et rendre effectif le parallélisme potentiel de données, il faut dans un premier temps construire explicitement ces listes de déplacements, puis les découper en parties égales afin de répéter l'opération autant de fois que nous avons de parties.

Nous avons donc créé une opération (Fig. 30) qui, à partir des deux couples de déplacements, génère deux listes contenant tous les déplacements et des sorties indiquant le nombre d'éléments de chacune des listes. Sur la figure 31 nous pouvons remarquer que nous avons découpé la liste des déplacements en ordonnée en quatre. Il apparaît donc quatre opérations.

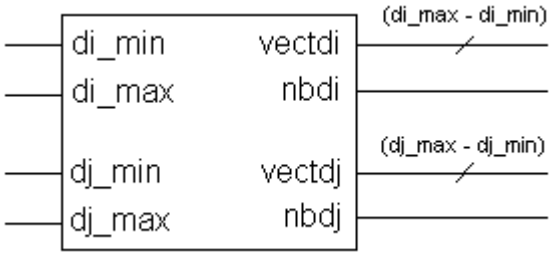


Figure 30 : Opération GenereVecteurIndices.

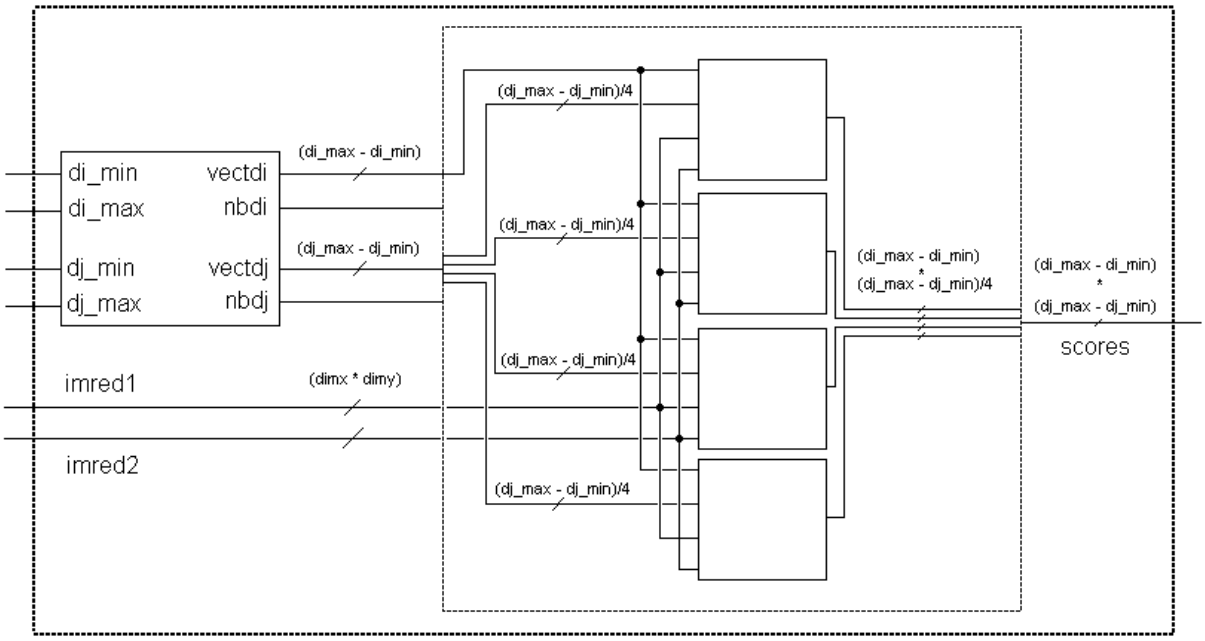


Figure 31 : Opération CorrelationsDeplacements avec du parallélisme de données.

Le cas `cond = 1` de l'opération `CorrelationsDeplacements` est composé des opérations :

- `MotifCorrelIm1`, qui permet d'extraire le motif de corrélation (`MCIm1`) correspondant au voisinage du centre de l'image réduite 1.
- `GenereVecteurIndices`, qui génère tous les déplacements à effectuer sous forme de listes (`vectdi` et `vectdj`) avec le nombre de déplacements pour chacune d'elles à partir des déplacements minimums (`di_min` et `dj_min`) et maximums (`di_max` et `dj_max`).

Chaque répétition de l'opération `CorrelationLigneP` récupère toute la liste des déplacements en abscisse et une partie différente de la liste des déplacements en ordonnée.

- `CorrelationLigneP`, qui effectue les calculs de corrélation entre le motif de corrélation et l'image réduite 2 selon les déplacements. Elle produit les scores de corrélation (`scores`) et les déplacements (`deplacs`) correspondants.
- `CoordonneesPicCorrel` récupère en entrée tous les scores (`scores`) et les déplacements (`deplacs`) correspondants issus des différentes répétitions de l'opération `CorrelationLigneP`. Afin de rechercher, parmi la liste des scores, le score minimum (`score`) et de déterminer les coordonnées du pic de corrélation (`res`) correspondant aux nouvelles coordonnées du centre de l'image réduite 1 sur l'image réduite 2.

5.2.4 Spécification hiérarchique de l'opération Appar

Dans l'opération `Appar`, nous pouvons faire apparaître du parallélisme de données en découpant la liste des points caractéristiques extraits de l'image 1 et en diffusant la liste des points caractéristiques extraits de l'image 2, de façon à appliquer la même opération (répétition de la fonction d'appariement) à chacune des parties de la liste des points caractéristiques extraits de l'image 1.

En regardant le programme C correspondant à notre algorithme, de calcul d'une translation entre deux images, nous nous apercevons que la partie correspondant à l'opération `Appar` est composée de plusieurs conditionnements de type « if(…)continue ».

Comme nous l'avons indiqué, au §5.2.2, si le conditionnement n'utilise aucun résultat de l'opération à répéter, nous plaçons le conditionnement dans une opération qui la précède. Dans notre application, l'élimination des points caractéristiques (de chacune des listes) se trouvant près des bords de l'image peut être effectuée avant l'opération d'appariement (opération à répéter). C'est pour cela que nous avons créé les opérations `ElimBord1` et `ElimBord2` (Fig. 32), qui suppriment, des deux listes de points caractéristiques, tous les points se trouvant à une distance inférieure à `halfwin` des bords de l'image.

Par contre si le conditionnement utilise les résultats de l'opération à répéter, nous effectuons un conditionnement « simple » (if(…)) et nous envoyons tous les résultats à une opération qui la suit en lui indiquant si chacun des résultats est correct ou non. L'opération placée à la sortie de l'opération répétée, fera une sélection des résultats à exploiter. Dans notre application, certains points ne sont pas appariés par l'opération `AppariementEtape1`, nous indiquons donc à l'opération chargée de la sélection des points appariés (`SelectionPtsAppar`) la validité des résultats, grâce à la valeur de `minscores` (si le minscore est égal à –1 l'appariement est erroné, sinon il représente le minscore du point apparié).

L'opération `Appar` peut se décomposer en 5 opérations (Fig. 32) :

- `CalculHalfwin`, qui calcule la valeur de la moitié du motif de corrélation à partir de la taille du motif de corrélation
- `ElimBord1` et `ElimBord2`, qui permettent d'éliminer les points se trouvant trop près des bords de l'image
- `AppariementEtape1`, qui permet d'apparier les points caractéristiques extraits de l'image 1 à ceux extraits de l'image 2 (opération à répéter).
- `SelectionPtsAppar`, qui trie les résultats provenant des différentes répétitions de l'opération `AppariementEtape1` pour déterminer la liste des points appariés.

Le premier niveau de spécification hiérarchique de l'opération `Appar` (Fig. 32) fait apparaître la répétition de l'opération `AppariementEtape1`. La répétition intervient sur la liste des points caractéristiques de l'image 1 (`plots_prev`).

Remarque :

La spécification hiérarchique de l'opération `CalculHalfwin` n'est pas représentée car elle correspond simplement une soustraction (de 1) suivie d'une division (par 2), qui permettent d'effectuer le calcul de `halfwin` à partir de la constante `WIN`.

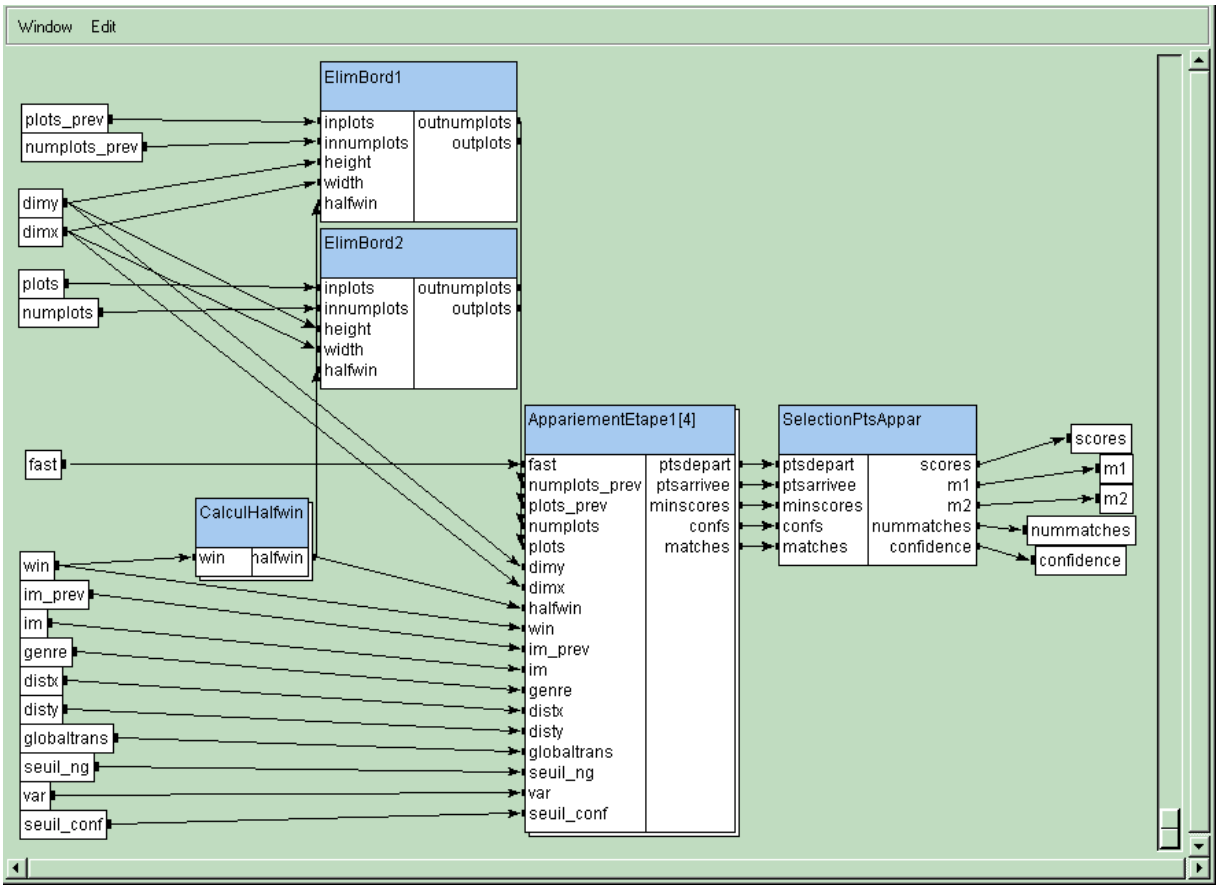


Figure 32 : Spécification hiérarchique de niveau 1 de l'opération Appar.

Dans l'opération `AppariementEtape1` nous pouvons faire apparaître du parallélisme de données en créant explicitement des sous-listes de points caractéristiques extraits de l'image 2 et en diffusant une partie (due à la répétition de l'opération `AppariementEtape1`) de la liste des points caractéristiques extraits de l'image 1.

Pour créer ces sous-listes, nous prenons chaque point caractéristique extrait de l'image 1 et nous lui associons une zone de recherche (déterminée en fonction de la translation globale estimée et de la position du point). Tous les points caractéristiques extraits de l'image 2 se trouvant dans une zone de recherche constitueront la sous-liste (des points caractéristiques de l'image 2) du point caractéristique extrait de l'image 1 associé à cette même zone de recherche.

L'opération d'appariement (`AppariementEtape2`, Fig. 33) est répétée, et effectue les calculs de corrélation sur différentes parties des sous-listes des points caractéristiques de l'image 2 (`sousplots`). L'opération répétée effectue les calculs de corrélation pour chacun des points de la sous-liste (des points caractéristiques de l'image 2) associée à chacun des points caractéristiques extraits de l'image 1. Chacun des scores de corrélation, correct ou non, est ensuite transmis à l'opération suivante qui effectue une sélection à partir des résultats et des indications fournies. L'opération `ScoreMinimumEtConfiance` récupère toutes les données produites par les différentes répétitions de l'opération `AppariementEtape2`, recherche dans chacune des sous-listes le point ayant le score minimum et calcul la confiance de l'appariement réalisé.

L'opération `AppariementEtape1` peut se décomposer en 3 opérations (Fig. 33) :

- `FenetragePtsCaracteristiques`, qui permet de sélectionner à partir d'un point de la liste des points caractéristiques extraits de l'image 1 et de la translation estimée globale, des points caractéristiques extraits de l'image 2 se trouvant dans la fenêtre de recherche. Ces points sélectionnés sont rangés sous forme de sous-listes.
- `AppariementEtape2`, qui permet d'effectuer tous les calculs de corrélation entre chacun des points caractéristiques extraits de l'image 1 et chacune des sous-listes (des points caractéristiques extrait de l'image 2) associée.
- `ScoreMinimumEtConfiance`, qui permet de trouver le score minimum contenu dans chacune des sous-listes, afin d'effectuer l'appariement et de calculer la confiance attribuée à cet appariement pour chacun des points caractéristiques extraits de l'image 1.

Le deuxième niveau de spécification hiérarchique de l'opération `Appar` (Fig. 33) fait apparaître la répétition de l'opération `AppariementEtape2`. La répétition intervient sur les sous-listes des points caractéristiques de l'image 2 (`sousplots`).

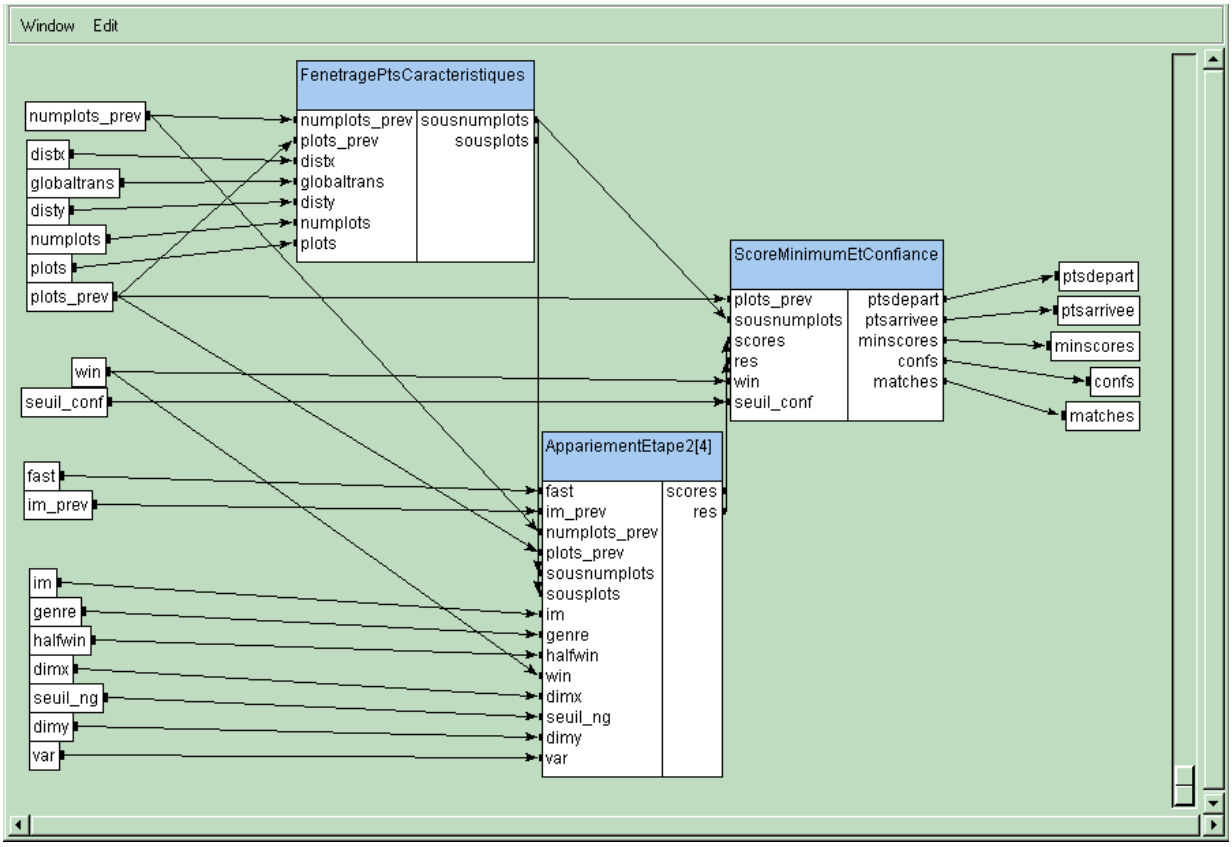


Figure 33 : Spécification hiérarchique de l'opération AppariementEtape1 (niveau 2 de Appar).

Les répétitions présentes sur les deux niveaux de la spécification hiérarchique de l'opération `Appar` apportent suffisamment de parallélisme potentiel de données, c'est pour cela que nous ne poursuivons pas cette spécification hiérarchique sur un troisième niveau.

5.2.5 Implantation de l'algorithme complet avec parallélisme de données

En appliquant la spécification hiérarchique de l'opération `Preestim` et la spécification hiérarchique de l'opération `Appar` à l'algorithme de calcul d'une translation entre deux images (Fig. 22), nous obtenons l'algorithme représenté sur la figure 34 avec des ombres sur les opérations `Preestim` et `Appar`.

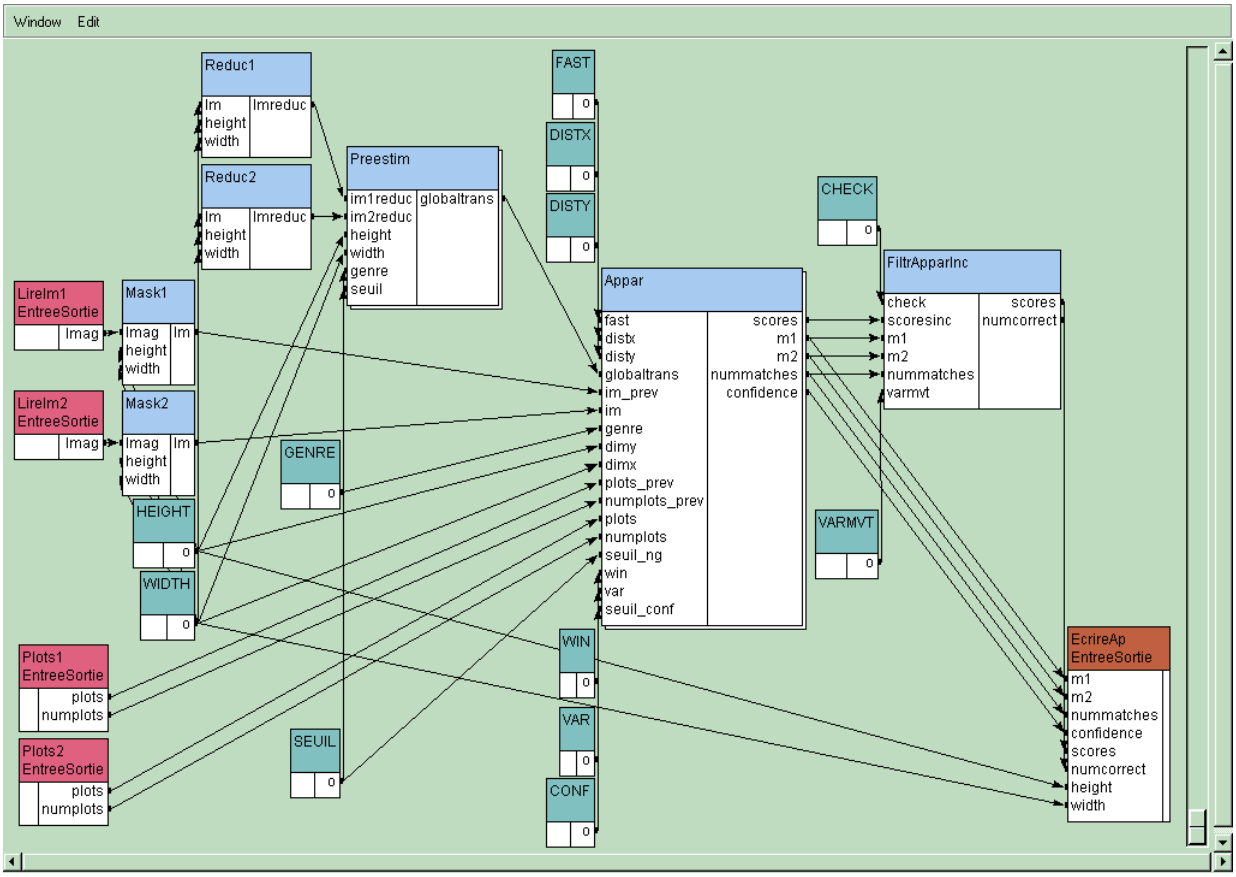


Figure 34 : Algorithme de calcul de translation entre deux images avec spécification hiérarchique.

Notre objectif est d'implanter l'algorithme de notre application (Fig. 34) sur une carte de traitement d'images. Cette carte est composée de processeurs de traitement de signal (DSP) de type ADSP21060. Ce processeur est produit par Analog Device, également connu sous le nom commercial « SHARC » pour (Super Harvard ARchitecture Computer). Afin de trouver implantation efficace de notre algorithme de calcul d'une translation entre deux images sur une architecture à base de ADSP21060, il faut chronométrer les différentes durées d'exécution des opérations (fonctions associées) sur ce processeur et réinjecter les valeurs trouvées dans SynDEx.

Les différentes fonctions associées aux opérations (Annexe 4) ont été chronométrées (Annexe 5) sur un des quatre ADSP21060 implantés sur une carte du commerce. Pour les besoins de la simulation avec le logiciel SynDEx v6, nous avons dû créer une librairie « Sharc » pour ce processeur. Elle contient non seulement les temps d'exécution des opérations de base mais aussi les temps de communication des liens point à point et des bus.

Sur la documentation du constructeur Analog Device, il est indiqué qu'une communication par lien point à point entre deux ADSP21060 est cadencée à 40Mo/s (25ns pour transmettre un octet) et que la communication sur un bus par le port externe est cadencée à 240Mo/s (6 fois plus rapide que le lien point à point, à 4,17 ns pour transmettre un octet).

Nous rencontrons un problème lorsque nous voulons injecter les durées d'exécution chronométrées des opérations pour le SHARC ADSP21060 dans SynDEx v6 (« Operators timings » dans l'item « Characteristics » du menu « Edit » dans le graphe d'architecture de notre application) et les temps de communication dans la librairie « sharc.sdx ». En effet, dans la version de SynDEx v6 dont nous disposons, la valeur maximale de la durée totale d'exécution de l'algorithme est limitée à environ 1.073.000.000 unités de temps. Pour avoir une bonne estimation de l'implantation, il faut maintenir la proportionnalité existante entre toutes les durées d'exécution des opérations et tous les temps de communication. Mais dans notre cas, nous ne pouvons pas respecter cette proportionnalité dans la mesure où nous sommes limités par la valeur maximale de la durée totale d'exécution de l'algorithme. La durée totale d'exécution de notre algorithme en monoprocesseur est de 30.103.739 µs (Cf. Annexe 5), et la durée de communication minimale est de 4,17 ns. Le rapport entre les deux est de 7.224.897.362, ce qui largement supérieur à la valeur dont nous disposons.

Pour avoir une implantation proche de l'implantation optimisée, nous devons rechercher le rapport maximum disponible, qui est le rapport entre la valeur maximale de la durée totale d'exécution (1.073.000.000) et la durée totale d'exécution de notre algorithme en monoprocesseur (30.103.739), c'est à dire environ 35 (35,64 précisément). Les nouvelles durées d'exécution à injecter dans SynDEx, avec le rapport de 35, sont disponibles en Annexe 5, l'unité de temps étant de 28,57 ns (1µs/35).

La plus petite unité de temps que nous pouvons avoir pour cette application est donc de 28,57 ns ; or nous avons des communications qui ont des temps inférieurs. Pour effectuer la simulation, nous fixons les temps de communication à cette unité de temps. Cela apparaît lors de la simulation, comme une utilisation de médias de communication plus lents. Selon le type de communication, la différence est plus ou moins importante. Ainsi, pour les liens point à point, le temps de communication est relativement proche (1,14 fois plus lent) de sa véritable valeur. Par contre, les communications par bus sont extrêmement ralenties (6,86 fois plus lent). Ces modifications sur les temps de communications entraînent une faible erreur sur le résultat de l'adéquation. Autant la différence pour les communications par liens point à point est négligeable autant elle ne l'est pas pour les communications par bus.

Pour estimer l'impact de l'implantation de l'algorithme de calcul de la translation entre deux images, sur la nouvelle version de SynDEx, nous allons effectuer quelques tests sur deux architectures (mono-processeur et bi-processeur) et utiliser ou non le parallélisme de données. Pour évaluer le gain apporté par chacune des implantations de notre algorithme sur des architectures différentes, il faut relever la durée d'exécution de notre algorithme et la comparer à la durée d'exécution de notre algorithme en mono-processeur. Pour cela, nous lançons l'adéquation de notre algorithme de calcul de translation entre deux images (Fig. 22) sur l'architecture mono-processeur (Fig. 35).

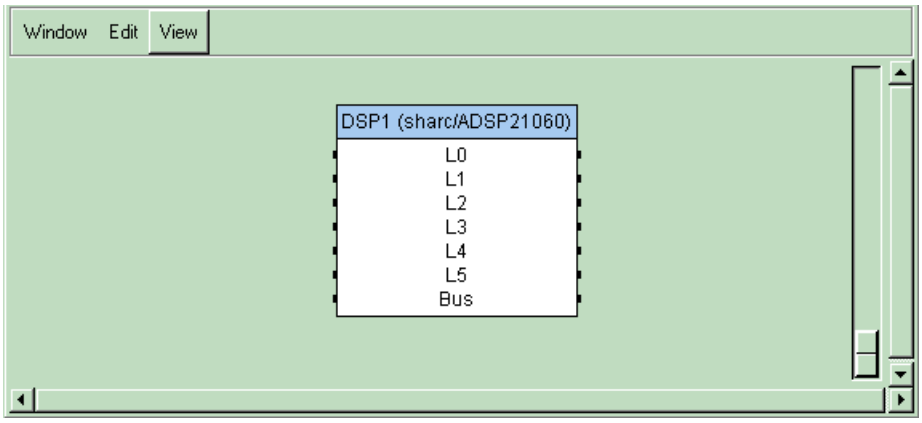


Figure 35 : Architecture mono-processeur (ADSP21060)

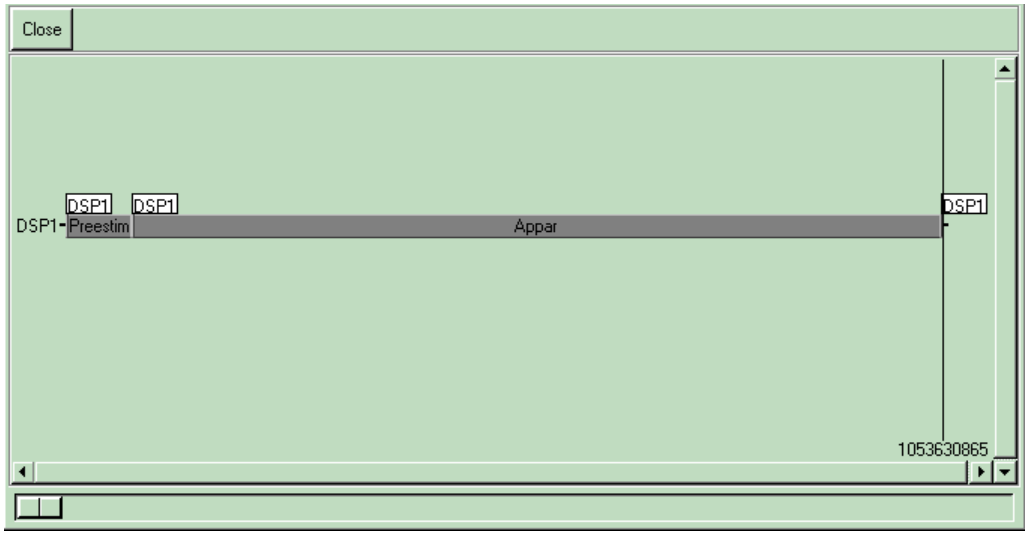


Figure 36 : Distribution/ordonnancement de l'algorithme complet en mono-processeur.

Le résultat de l'adéquation est un ordonnancement seulement sans distribution (cas mono-processeur), et nous relevons la durée d'exécution (Fig. 36) qui est de 1.053.630.865 unités de temps. Une fois cette valeur trouvée, nous allons déterminer le gain sur la durée d'exécution de l'algorithme apporté par une architecture bi-processeur (Fig. 37) sans parallélisme de données (Fig. 38), puis avec parallélisme de données (Fig. 39).

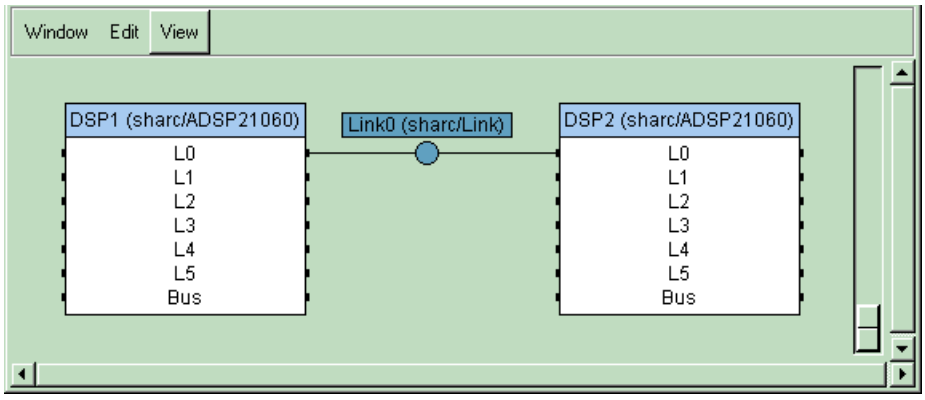


Figure 37 : Architecture bi-processeur (ADSP21060).

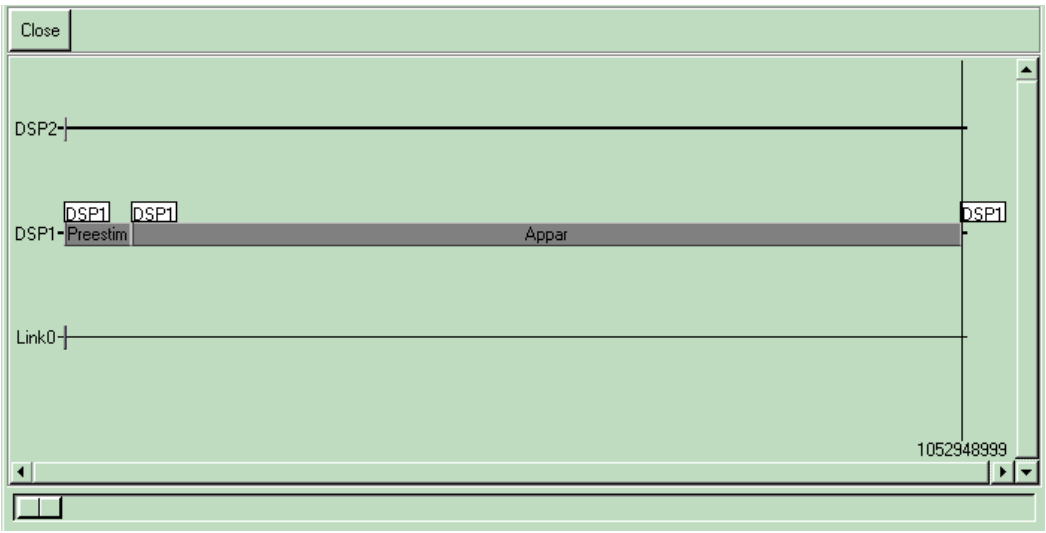


Figure 38 : Distribution/ordonnancement de l'algorithme en bi-processeur sans parallélisme de données.

On voit sur la figure 38 que la fonction `Preestim` utilise des données produites par les opérations `Mask2` et `Reduc2` exécutées par le processeur `DSP2`. La durée totale d'exécution de notre algorithme sans parallélisme de données sur une architecture bi-processeur (Fig. 37) est de 1.052.948.999 unités de temps (Fig. 38). Nous pouvons remarquer que sans le parallélisme de données le processeur `DSP2` n'est que très peu utilisé. Le gain apporté par cette architecture sans parallélisme de données est de 0,65% ; ce qui est très faible pour l'ajout d'un nouveau processeur.

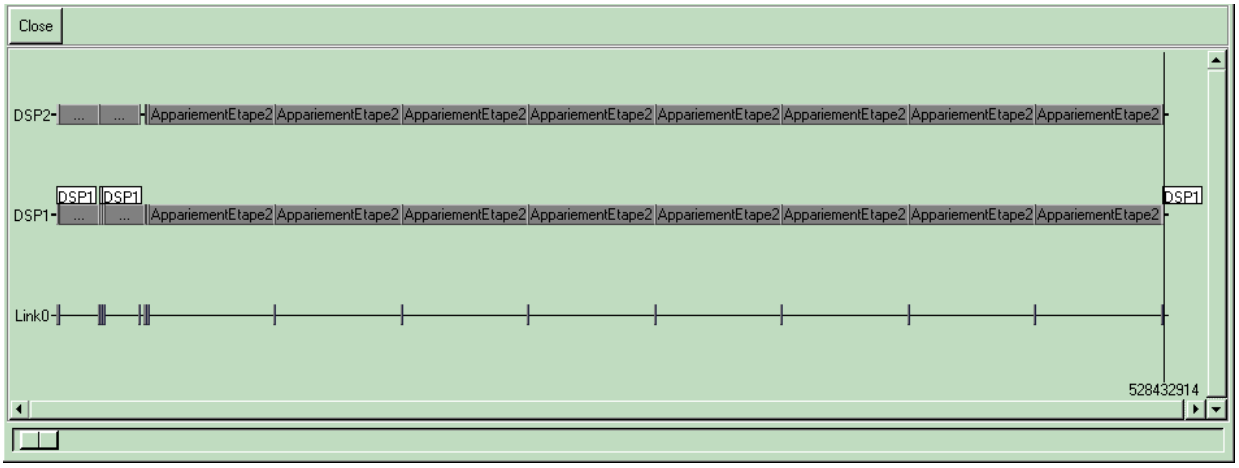


Figure 39 : Distribution/ordonnancement de l'algorithme en bi-processeur avec parallélisme de données.

L'utilisation du parallélisme de données à permis de réduire de façon conséquente la durée totale d'exécution de notre algorithme qui atteint 528.432.914 unités de temps (Fig. 39). Nous remarquons qu'avec le parallélisme de données le processeur `DSP2` est à présent très utilisé. Le gain apporté par cette architecture avec parallélisme de données est de 49,85% ; ce qui est un résultat très intéressant.

Avec notre application, nous avons montré, que la spécification hiérarchique de notre algorithme faisant intervenir du parallélisme potentiel permet de réduire significativement la durée totale d'exécution en multi-processeurs.

5.3 Exploration des implantations de l'algorithme sur différentes architectures

Nous avons vu précédemment que l'utilisation du parallélisme de données permet de réduire les durées d'exécution de l'algorithme. Nous allons explorer des implantations de notre algorithme sur différentes architectures et topologies afin de trouver le meilleur type d'architecture et d'implantation.

5.3.1 Présentation des différentes architectures

Une architecture est définie par son nombre de processeurs et par la topologie d'intersection entre ces processeurs. Nous avons choisi trois types de topologies pour nos implantations :

- point à point en anneau,
- point à point complètement connecté,
- point à point en anneau, avec un bus de communication.

Nous utilisons l'ADSP21060 qui possède 6 liens de communication point à point qui permettent de le relier à 6 autres Sharc. Il possède aussi un port externe qui permet de le relier à un bus système sur lequel peuvent être connectés d'autres Sharc et de la mémoire partagée.

A première vue, la topologie point à point en anneau est simple à réaliser techniquement (facile à mettre en œuvre), mais les durées de communications dépendent du nombre de liens intermédiaires.
La topologie point à point complètement connecté est difficile à mettre en œuvre car il y a énormément de liens, même s'ils sont tous connectés entre eux et que les durées de communications sont constantes et minimales.
La topologie point à point anneau + bus est une amélioration de la topologie point à point en anneau. Elle allie une certaine rapidité et une certaine facilité.

Nous avons choisi de faire les implantations de notre algorithme avec un nombre de processeur allant de trois à huit.
Pour chacune des implantations de notre algorithme sur une architecture différente, nous relevons les durées totales d'exécution de notre algorithme et nous calculons le gain apporté. Tous ces résultats sont rassemblés dans un tableau pour chacune des architectures.

3 processeurs

Lorsque nous avons 3 processeurs, la topologie point à point en anneau et celle point à point complètement connecté sont équivalentes. Donc nous ne lançons l'adéquation, de notre algorithme sur une architecture à trois processeurs, que sur la topologie point à point en anneau (Fig. 40) et celle point à point en anneau + bus (Fig. 42).

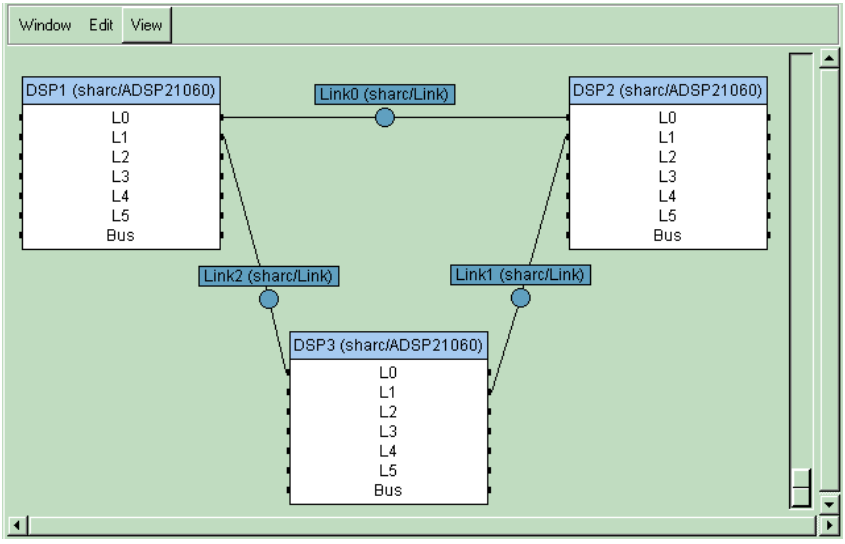


Figure 40 : Architecture à 3 processeurs, avec topologie point à point en anneau.

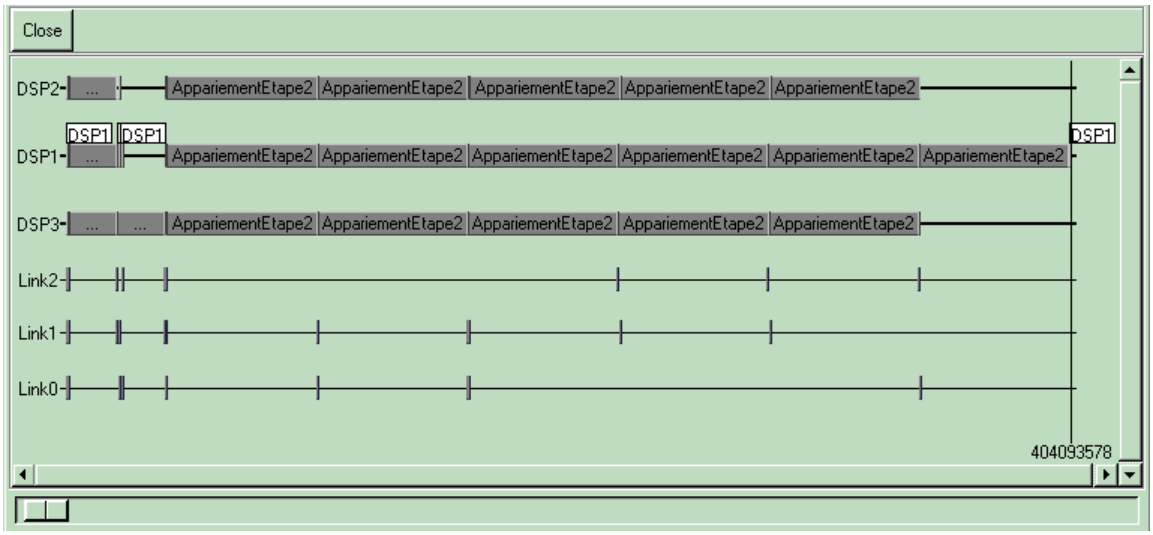


Figure 41 : Distribution/ordonnancement de l'algorithme sur l'architecture à 3 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à trois processeurs avec une topologie point à point en anneau (Fig. 40) est de 404.093.578 unités de temps (Fig. 41). Nous remarquons, par la présence d'espaces à la fin de l'ordonnancement des processeurs `DSP2` et `DSP3` (Fig. 41) que ces derniers ne sont pas exploités au maximum.

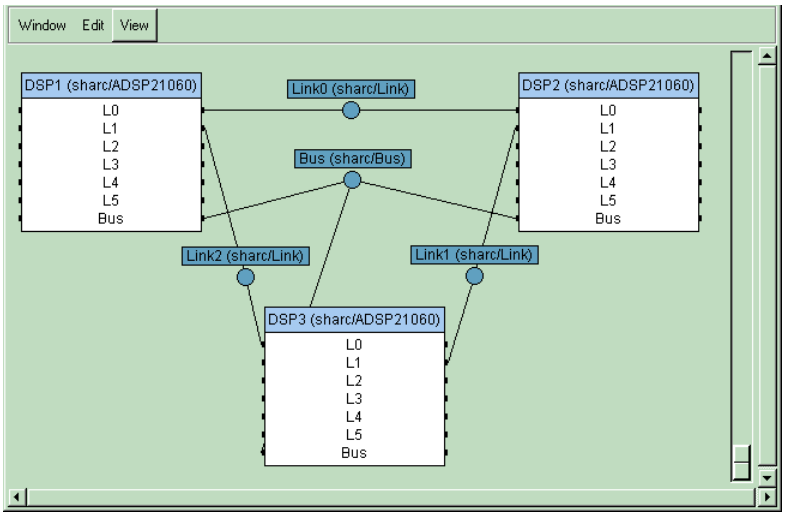


Figure 42 : Architecture à 3 processeurs, avec topologie point à point en anneau +bus.

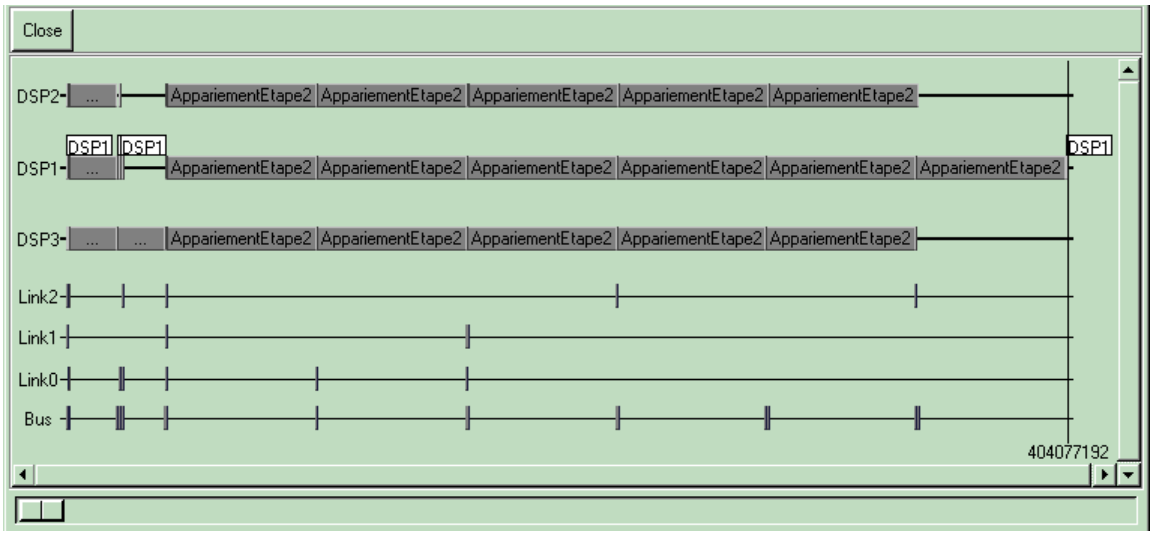


Figure 43 : Distribution/ordonnancement de l'algorithme sur l'architecture à 3 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à trois processeurs avec une topologie point à point en anneau avec un bus (Fig. 42) est de 404.077.192 unités de temps (Fig. 43). Nous remarquons, par la présence d'espaces dans les différentes distributions/ordonnancements (Fig. 39, Fig. 41 et Fig. 43) que les implantations de notre algorithme sur cette architecture ne sont pas optimisées.

Nbre de processeurs	3	
Topologie	Pt à pt, en anneau	Anneau + Bus
Durée (unités de tps)	404093578	404077192
Gain (en %)	61,648	61,649

Les gains apportés par les deux topologies sont très proches. Mais il ne faut pas oublier que les temps de communication du bus ont été multipliés par 6,86, ce qui modifie le résultat.

4 processeurs

Nous lançons l'adéquation, de notre algorithme sur une architecture à quatre processeurs, sur la topologie point à point en anneau (Fig. 44), la topologie point à point complètement connecté (Fig. 46) et la topologie point à point en anneau + bus (Fig. 48).

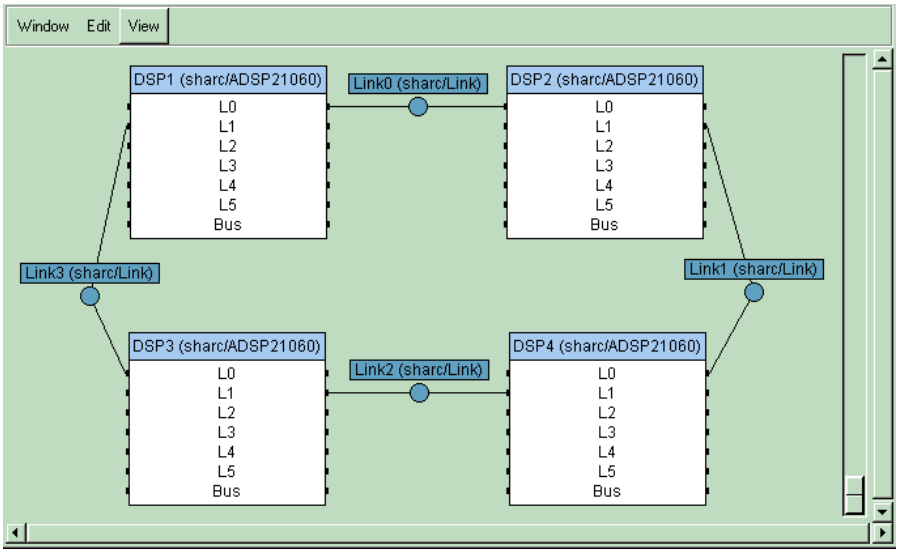


Figure 44 : Architecture à 4 processeurs, avec topologie point à point en anneau.

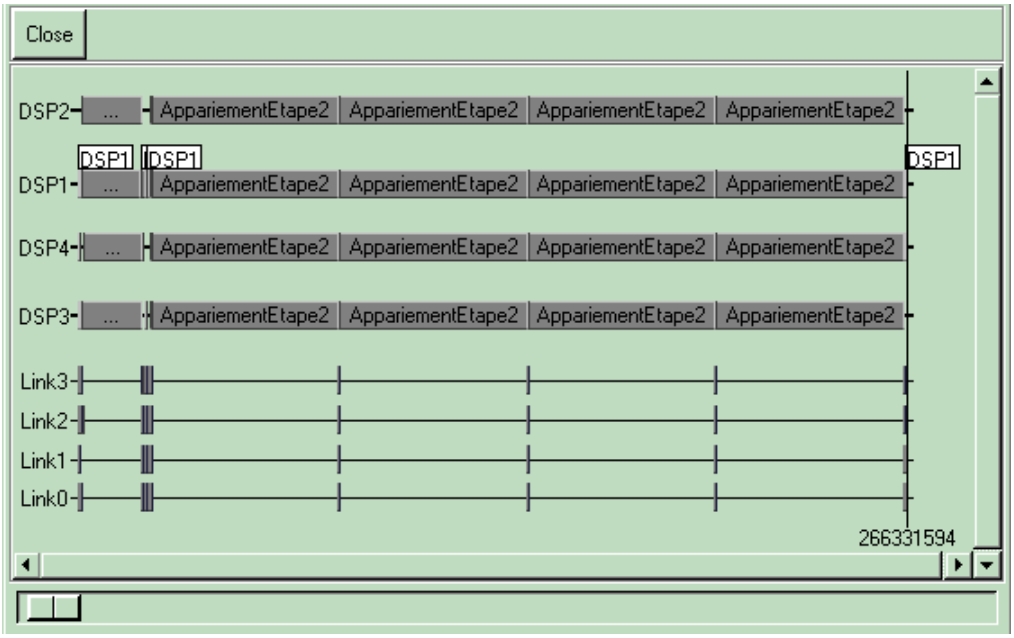


Figure 45 : Distribution/ordonnancement de l'algorithme sur l'architecture à 4 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à quatre processeurs avec une topologie point à point en anneau (Fig. 44) est de 266.331.594 unités de temps (Fig. 45). Nous remarquons sur la figure 45, par l'absence d'espace, que tous les processeurs sont très bien exploités.

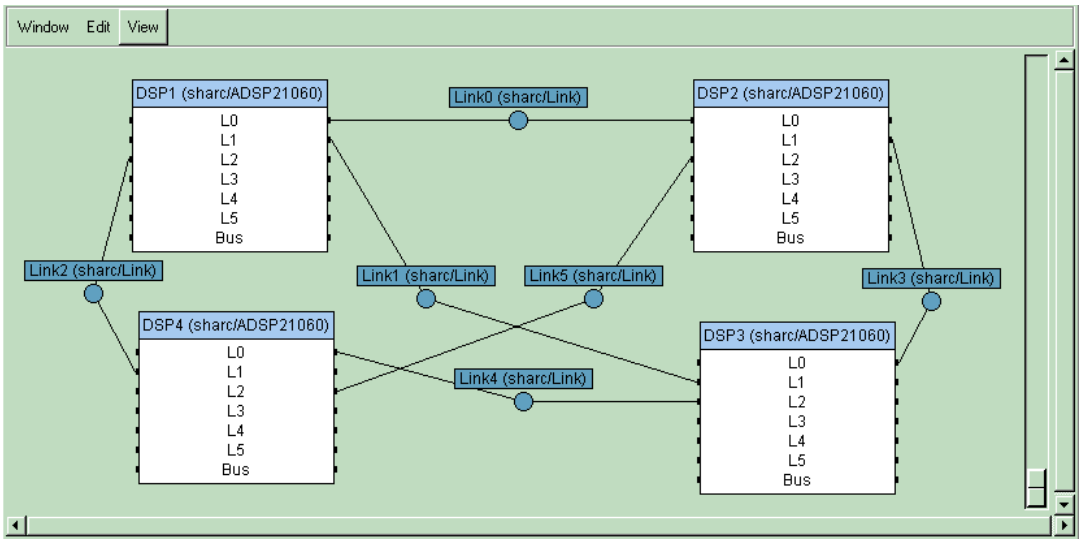


Figure 46 : Architecture à 4 processeurs, avec topologie point à point complètement connecté.

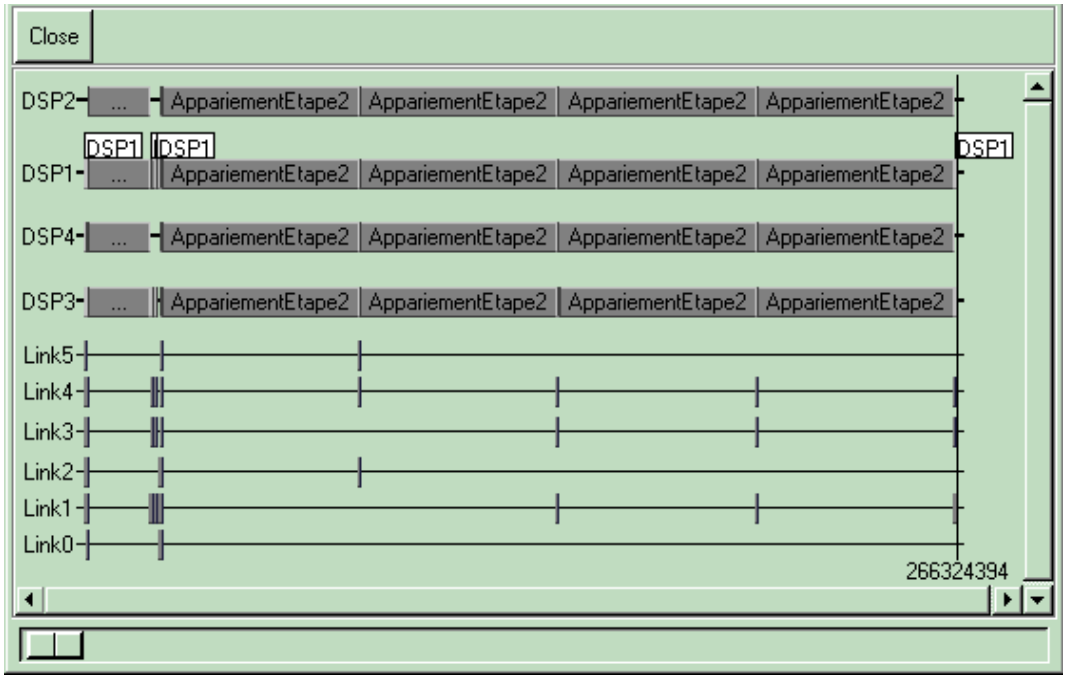


Figure 47 : Distribution/ordonnancement de l'algorithme sur l'architecture à 4 processeurs, avec topologie point à point complètement connecté.

La durée totale d'exécution de notre algorithme sur une architecture à quatre processeurs avec une topologie point à point complètement connecté (Fig. 46) est de 266.324.394 unités de temps (Fig. 47). Nous remarquons sur la figure 47, par l'absence d'espace, que tous les processeurs sont très bien exploités.

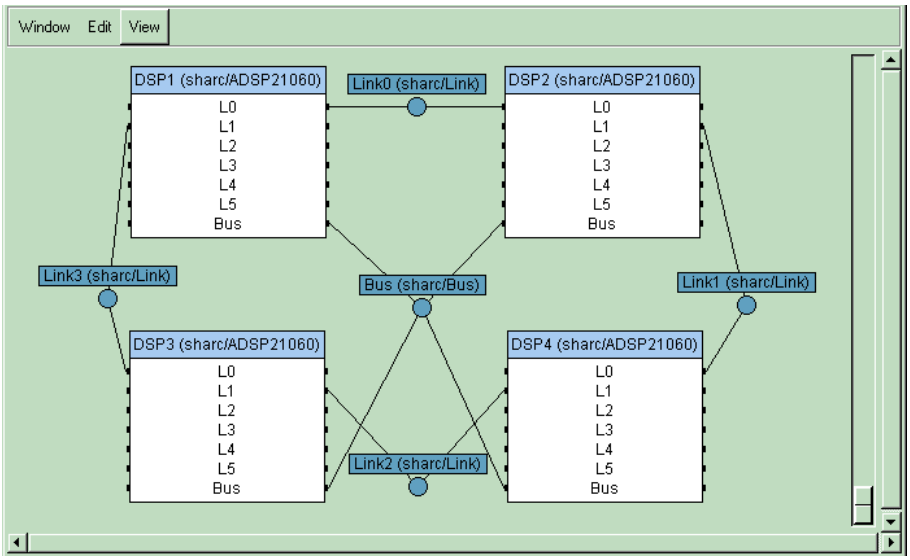


Figure 48 : Architecture à 4 processeurs, avec topologie point à point en anneau +bus.

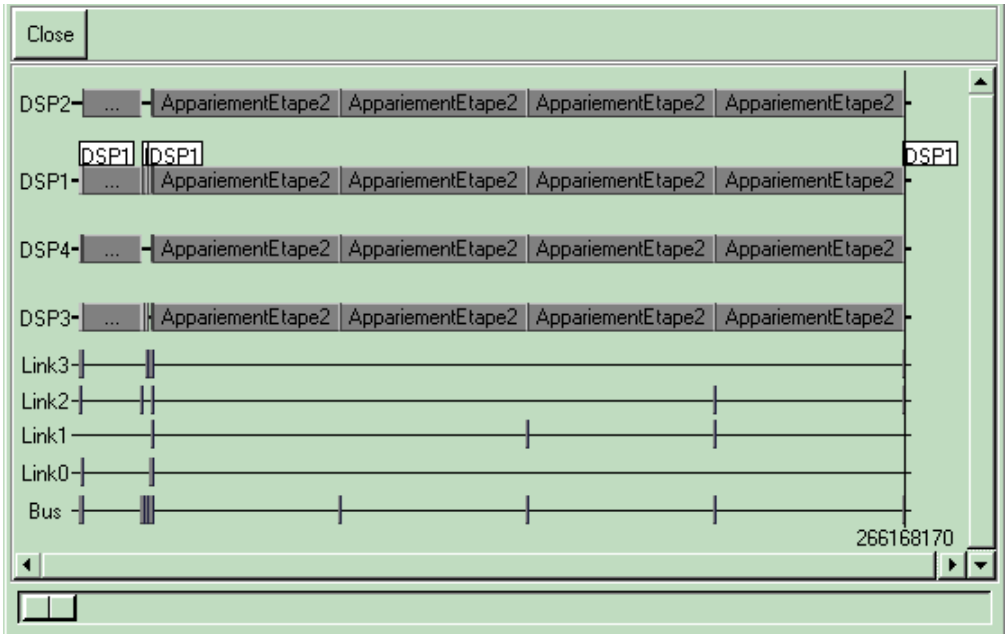


Figure 49 : Distribution/ordonnancement de l'algorithme sur l'architecture à 4 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à quatre processeurs avec une topologie point à point en anneau avec un bus (Fig. 48) est de 266.168.170 unités de temps (Fig. 49). Nous remarquons, par l'absence d'espaces dans les différentes distributions/ordonnancements (Fig. 45, Fig. 47 et Fig. 49) que les implantations de notre algorithme sur cette architecture sont optimisées.

Nbre de processeurs	4		
Topologie	Pt à pt, en anneau	Pt à pt, compl. connecté	Anneau + Bus
Durée (unités de tps)	266331594	266324394	266168170
Gain (en %)	74,722	74,723	74,738

5 processeurs

Nous lançons l'adéquation, de notre algorithme sur une architecture à cinq processeurs, sur la topologie point à point en anneau (Fig. 50), la topologie point à point complètement connecté (Fig. 52) et la topologie point à point en anneau + bus (Fig. 54).

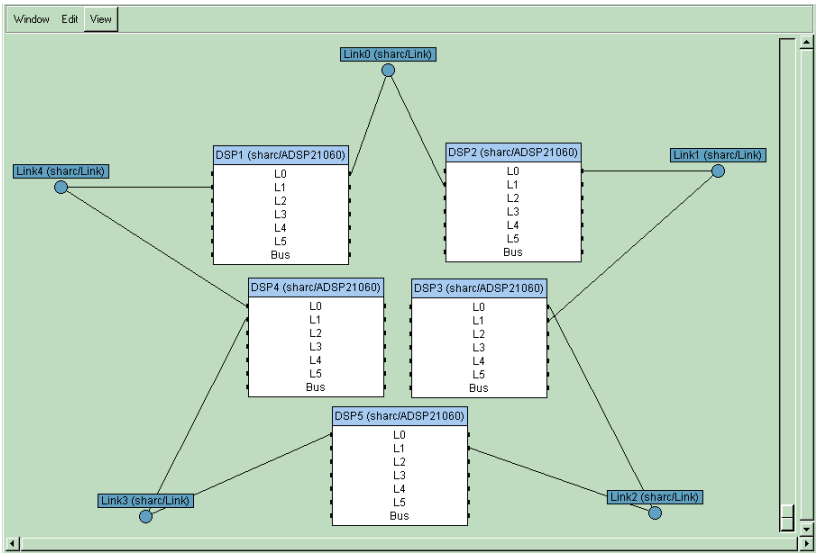


Figure 50 : Architecture à 5 processeurs, avec topologie point à point en anneau.

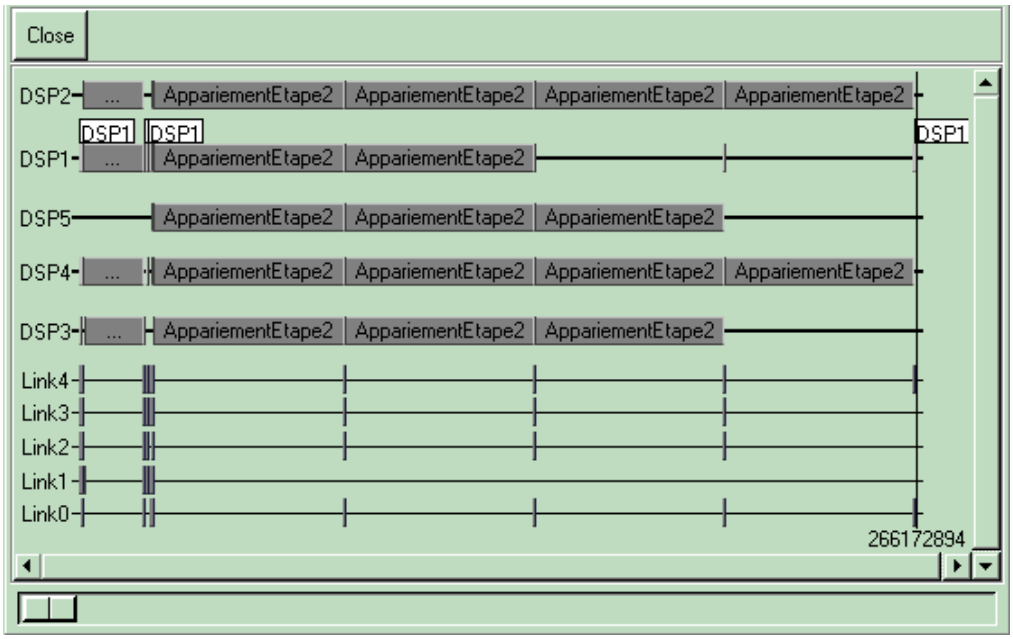


Figure 51 : Distribution/ordonnancement de l'algorithme sur l'architecture à 5 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à cinq processeurs avec une topologie point à point en anneau (Fig. 50) est de 266.172.894 unités de temps (Fig. 51). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP1`, `DSP3` et `DSP5` (Fig. 51) que ces derniers ne sont pas exploités au maximum.

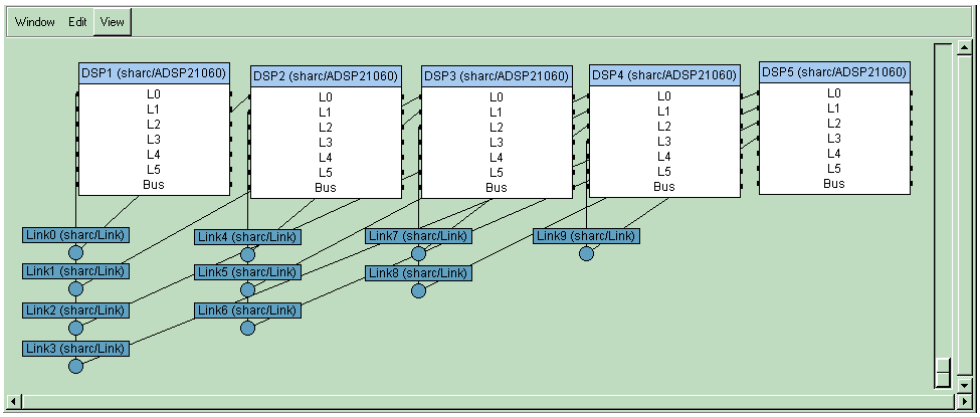


Figure 52 : Architecture à 5 processeurs, avec topologie point à point complètement connecté.

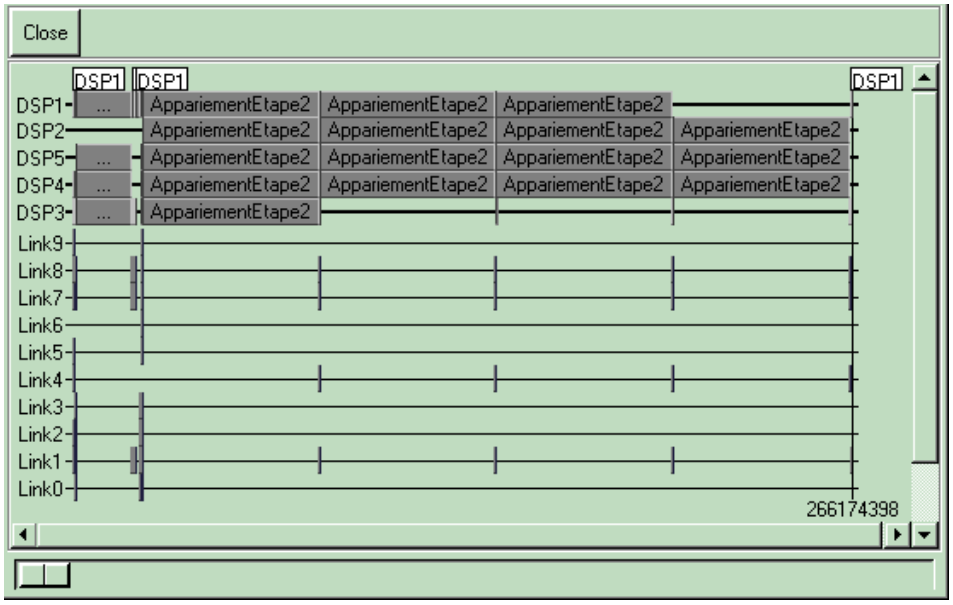


Figure 53 : Distribution/ordonnancement de l'algorithme sur l'architecture à 5 processeurs, avec topologie point à point complètement connecté.

La durée totale d'exécution de notre algorithme sur une architecture à quatre processeurs avec une topologie point à point complètement connecté (Fig. 52) est de 266.174.398 unités de temps (Fig. 53). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP1`, `DSP2` et `DSP3` (Fig. 53) que ces derniers ne sont pas exploités au maximum.

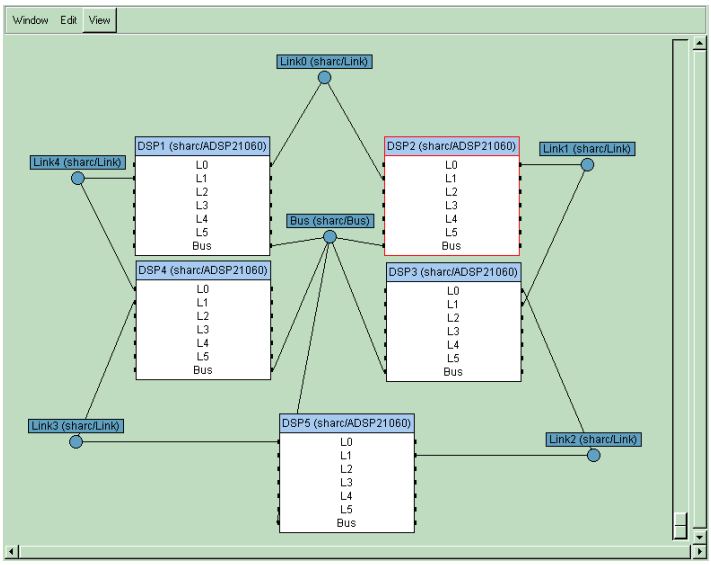


Figure 54 : Architecture à 5 processeurs, avec topologie point à point en anneau +bus.

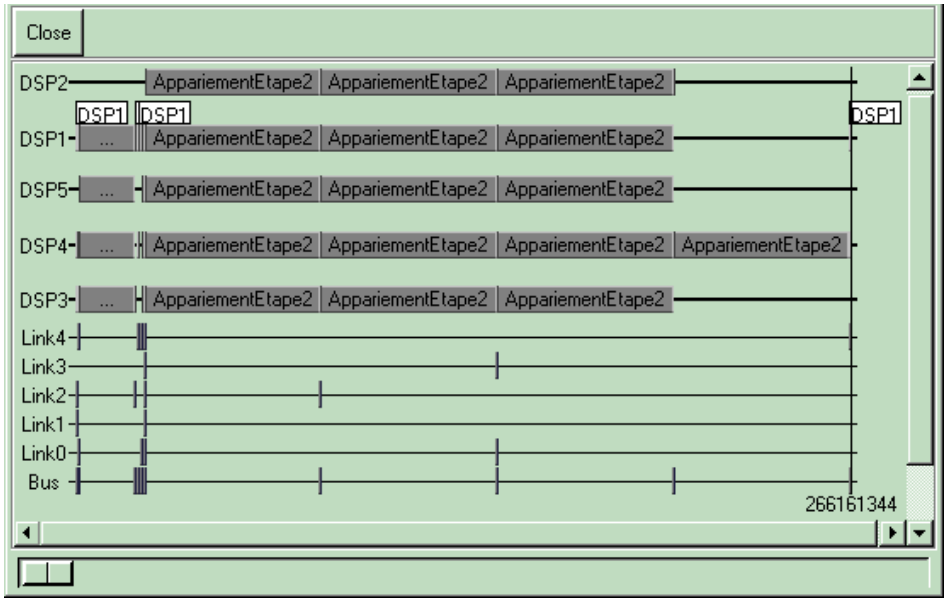


Figure 55 : Distribution/ordonnancement de l'algorithme sur l'architecture à 5 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à cinq processeurs avec une topologie point à point en anneau avec un bus (Fig. 54) est de 266.161.344 unités de temps (Fig. 55). Nous remarquons, par la présence d'espaces dans les différentes distributions/ordonnancements (Fig. 51, Fig. 53 et Fig. 55) que les implantations de notre algorithme sur cette architecture ne sont pas optimisées.

Nbre de processeurs	5		
Topologie	Pt à pt, en anneau	Pt à pt, compl. connecté	Anneau + Bus
Durée (unités de tps)	266172894	266174398	266161344
Gain (en %)	74,738	74,737	74,739

6 processeurs

Nous lançons l'adéquation, de notre algorithme sur une architecture à six processeurs, sur la topologie point à point en anneau (Fig. 56), la topologie point à point complètement connecté (Fig. 58) et la topologie point à point en anneau + bus (Fig. 60).

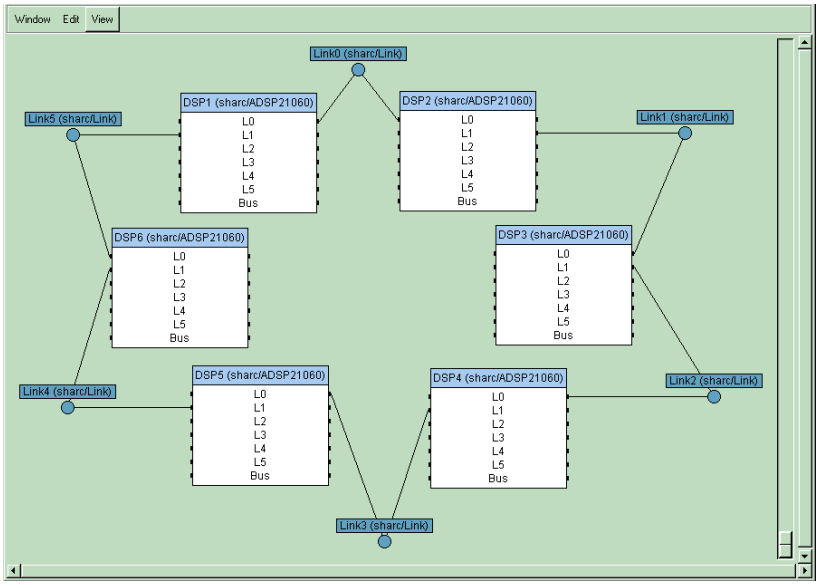


Figure 56 : Architecture à 6 processeurs avec topologie point à point en anneau.

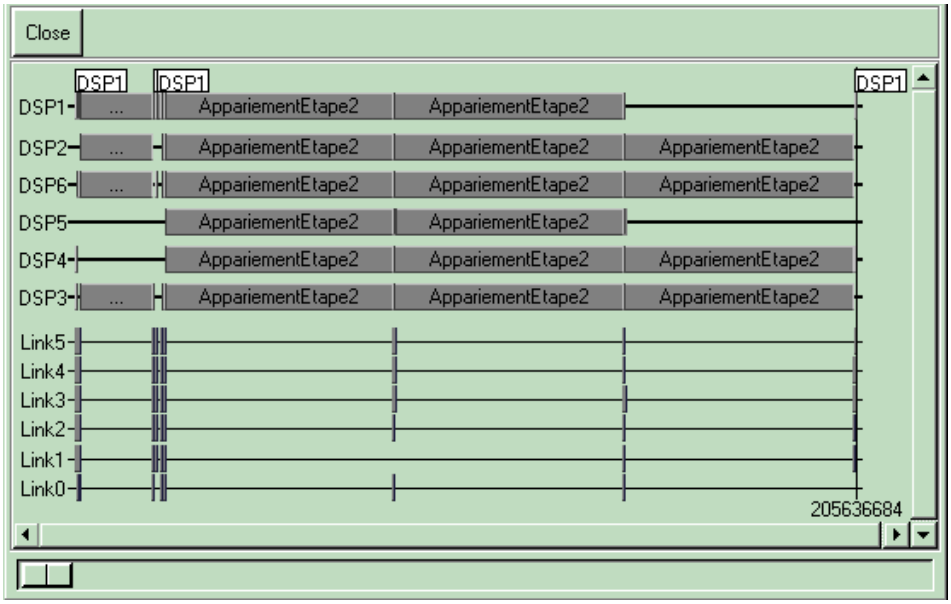


Figure 57 : Distribution/ordonnancement de l'algorithme sur l'architecture à 6 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à six processeurs avec une topologie point à point en anneau (Fig. 56) est de 205.636.684 unités de temps (Fig. 57). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP1`, `DSP4` et `DSP5` (Fig. 57) que ces derniers ne sont pas exploités au maximum.

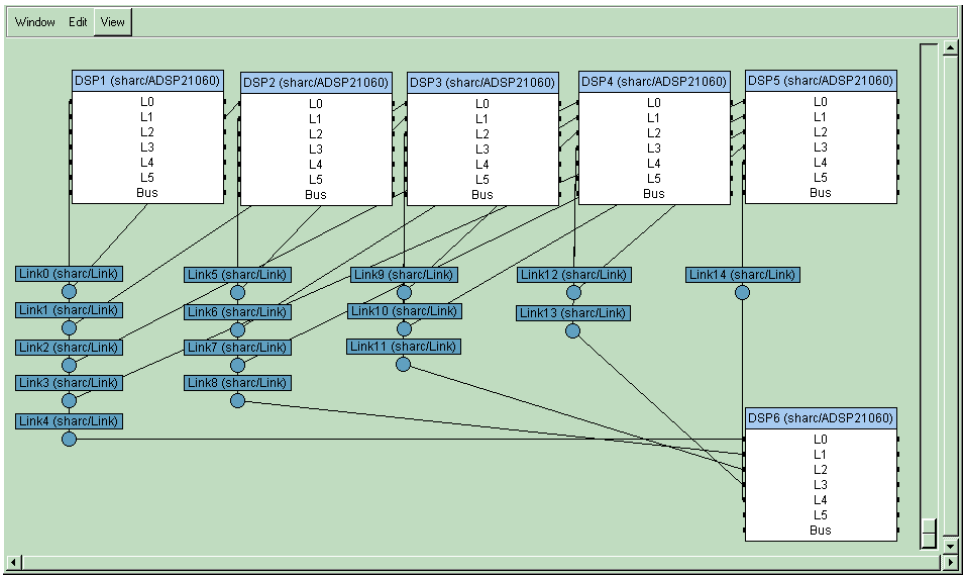


Figure 58 : Architecture à 6 processeurs, avec topologie point à point complètement connecté.

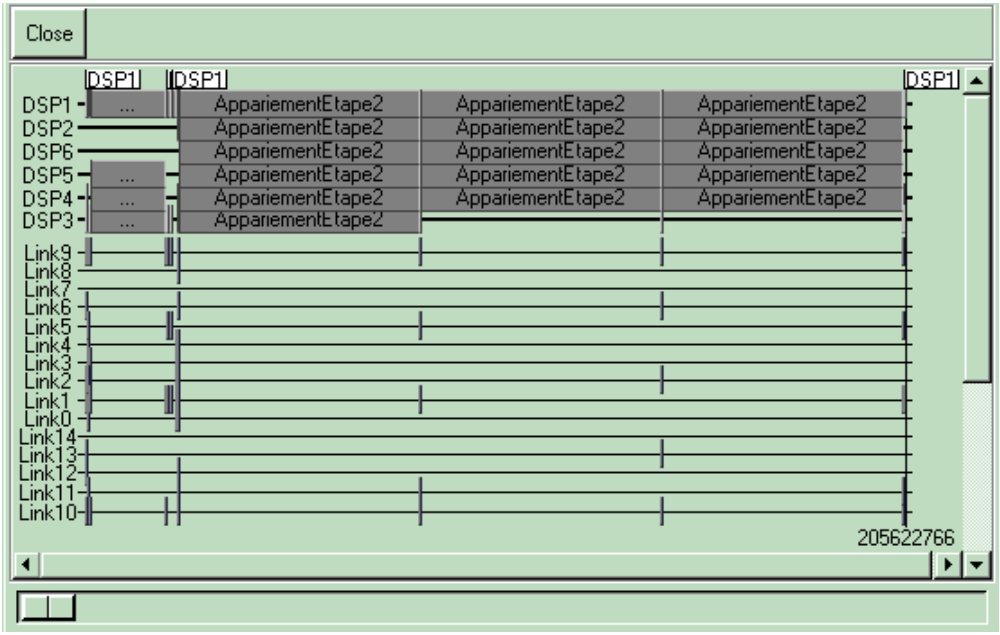


Figure 59 : Distribution/ordonnancement de l'algorithme sur l'architecture à 6 processeurs, avec topologie point à point complètement connecté.

La durée totale d'exécution de notre algorithme sur une architecture à quatre processeurs avec une topologie point à point complètement connecté (Fig. 58) est de 205.622.766 unités de temps (Fig. 59). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP2`, `DSP3` et `DSP6` (Fig. 59) que ces derniers ne sont pas exploités au maximum.

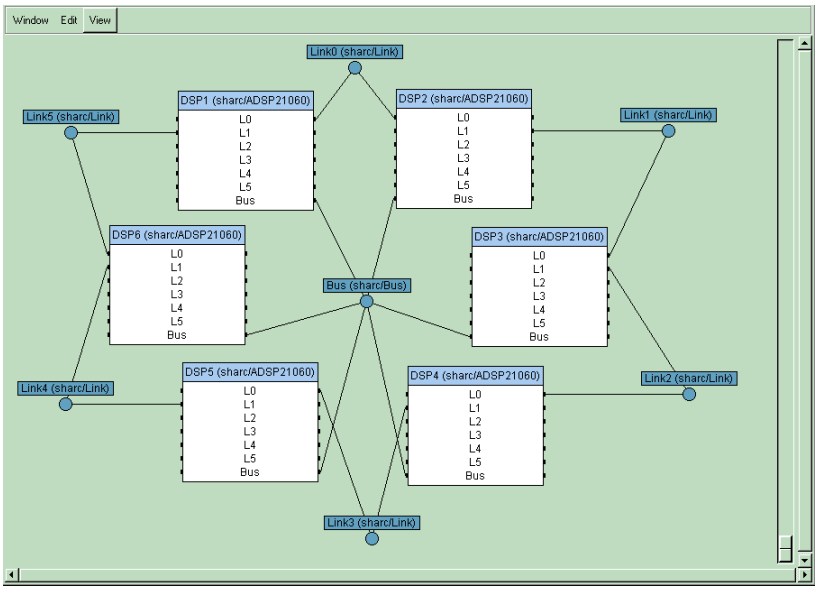


Figure 60 : Architecture à 6 processeurs, avec topologie point à point en anneau +bus.

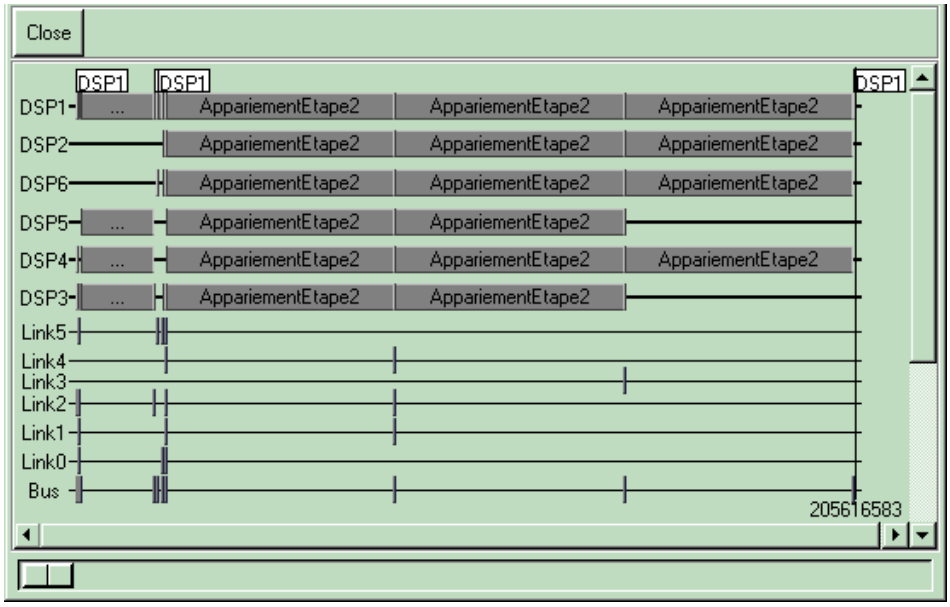


Figure 61 : Distribution/ordonnancement de l'algorithme sur l'architecture à 6 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à six processeurs avec une topologie point à point en anneau avec un bus (Fig. 60) est de 205.616.583 unités de temps (Fig. 61). Nous remarquons, par la présence d'espaces dans les différentes distributions/ordonnancements (Fig. 57, Fig. 59 et Fig. 61) que les implantations de notre algorithme sur cette architecture ne sont pas optimisées.

Nbre de processeurs	6		
Topologie	Pt à pt, en anneau	Pt à pt, compl. connecté	Anneau + Bus
Durée (unités de tps)	205636684	205622766	205616583
Gain (en %)	80,483	80,484	80,485

7 processeurs

Les liens de communication point à point de l'ADSP21060 ne permettent de relier un processeur qu'à cinq autres au maximum. Donc nous ne lançons l'adéquation, de notre algorithme sur une architecture à sept processeurs, que sur la topologie point à point en anneau (Fig. 62) et la topologie point à point en anneau + bus (Fig. 64).

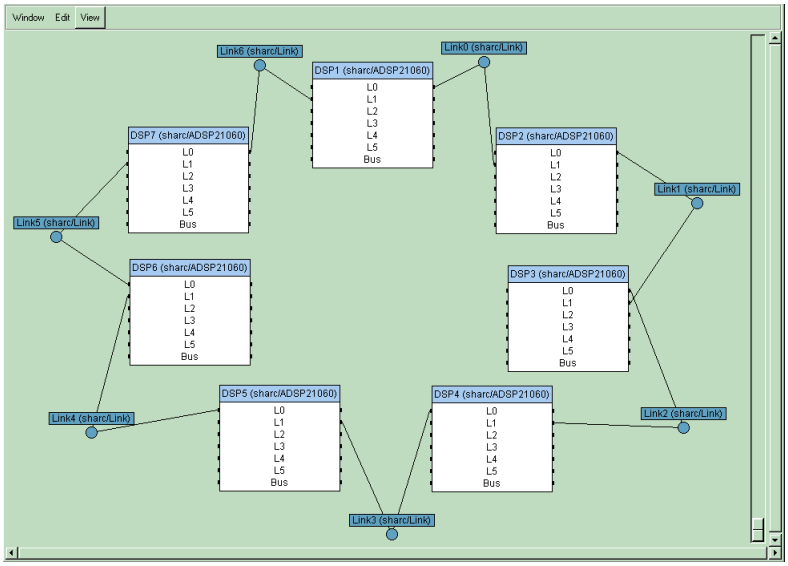


Figure 62 : Architecture à 7 processeurs avec topologie point à point en anneau.

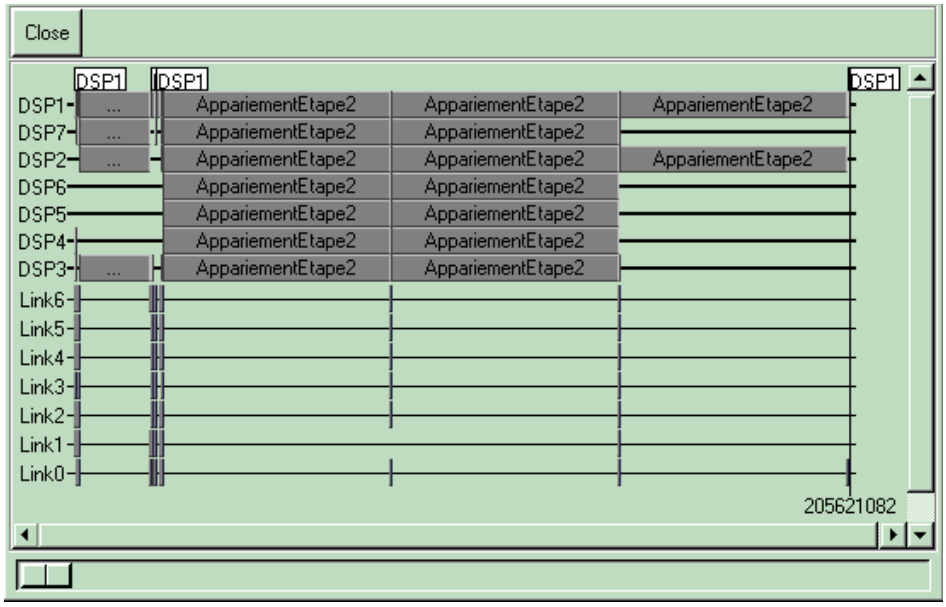


Figure 63 : Distribution/ordonnancement de l'algorithme sur l'architecture à 7 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à sept processeurs avec une topologie point à point en anneau (Fig. 62) est de 205.621.082 unités de temps (Fig. 63). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP3`, `DSP4`, `DSP5`, `DSP6` et `DSP7` (Fig. 63) que ces derniers ne sont pas exploités au maximum.

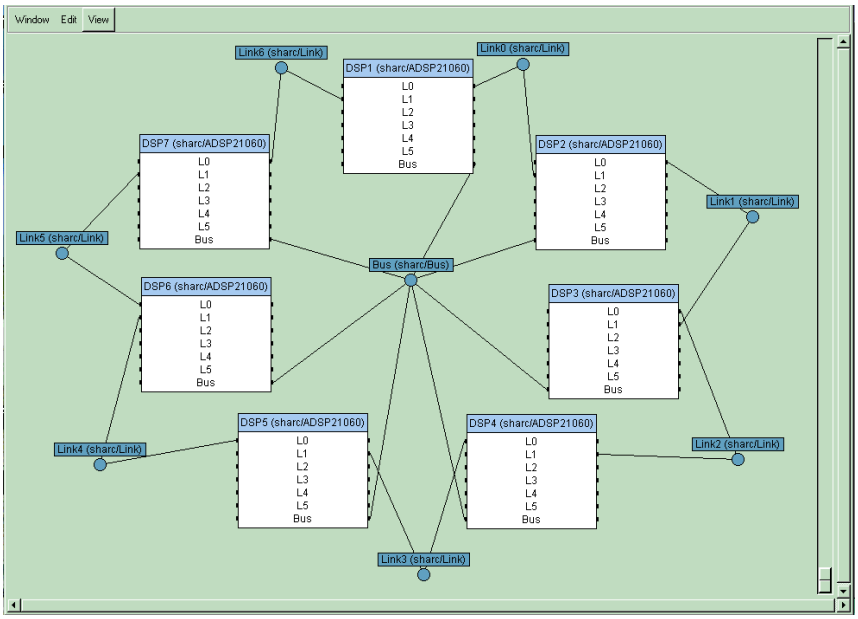


Figure 64 : Architecture à 7 processeurs, avec topologie point à point en anneau +bus.

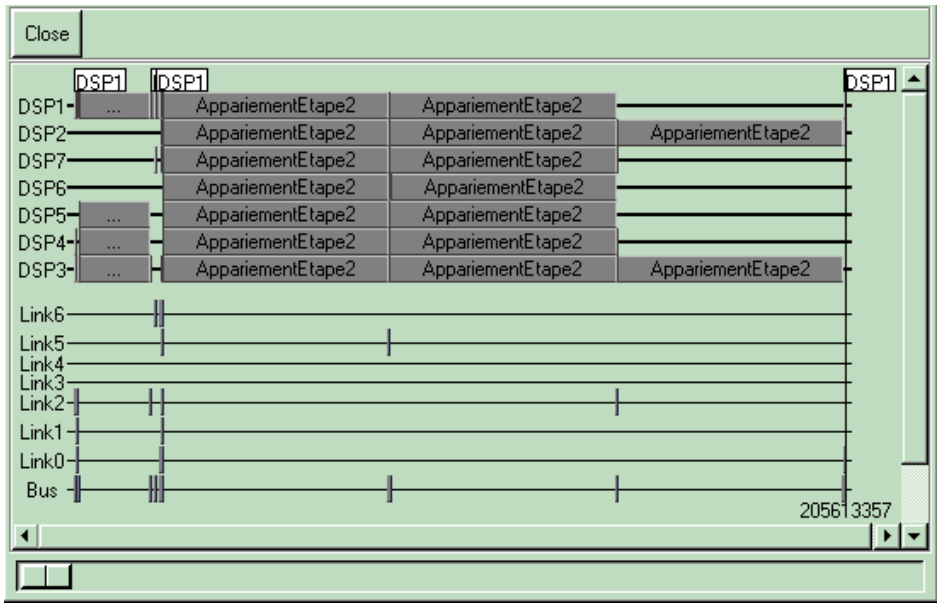


Figure 65 : Distribution/ordonnancement de l'algorithme sur l'architecture à 7 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à sept processeurs avec une topologie point à point en anneau avec un bus (Fig. 64) est de 205.613.357 unités de temps (Fig. 65). Nous remarquons, par la présence d'espaces dans les différentes distributions/ordonnancements (Fig. 63 et Fig. 65) que les implantations de notre algorithme sur cette architecture ne sont pas optimisées.

Nbre de processeurs	7	
Topologie	Pt à pt, en anneau	Anneau + Bus
Durée (unités de tps)	205621082	205613357
Gain (en %)	80,485	80,486

8 processeurs

Nous lançons l'adéquation, de notre algorithme sur une architecture à sept processeurs, sur la topologie point à point en anneau (Fig. 66) et la topologie point à point en anneau + bus (Fig. 68).

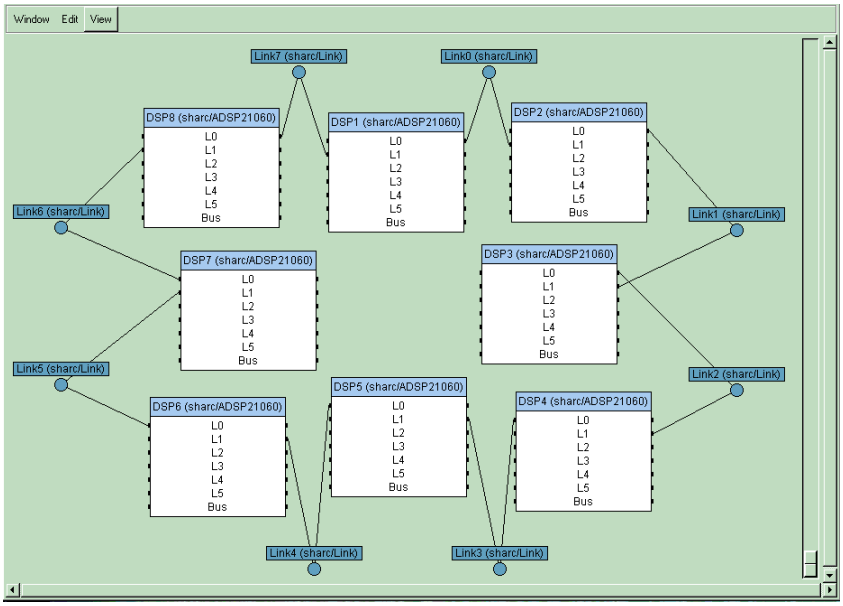


Figure 66 : Architecture à 8 processeurs avec topologie point à point en anneau.

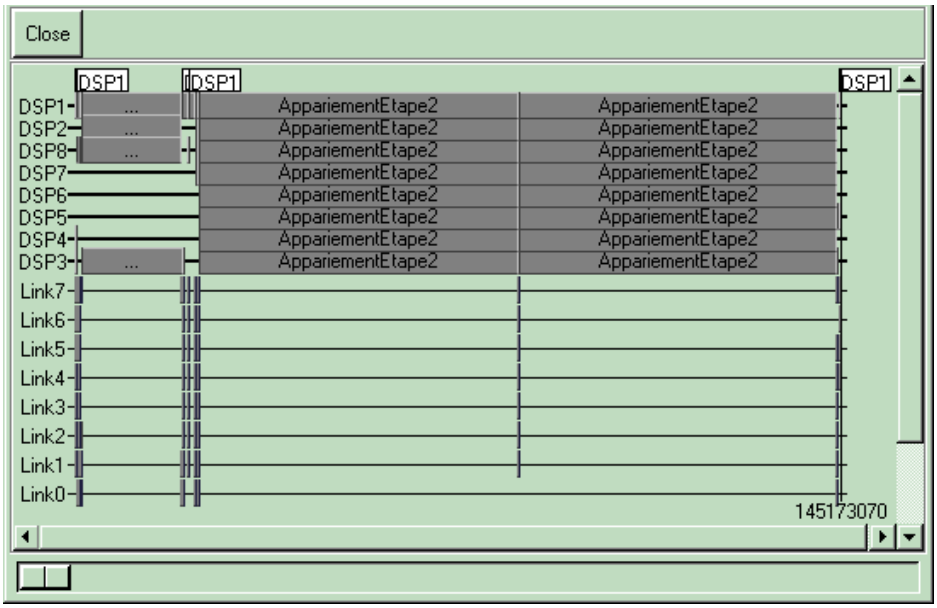


Figure 67 : Distribution/ordonnancement de l'algorithme sur l'architecture à 8 processeurs, avec topologie point à point en anneau.

La durée totale d'exécution de notre algorithme sur une architecture à huit processeurs avec une topologie point à point en anneau (Fig. 66) est de 145.173.070 unités de temps (Fig. 67). Nous remarquons, par la présence d'espaces dans l'ordonnancement des processeurs `DSP4`, `DSP5`, `DSP6` et `DSP7` (Fig. 67) que ces derniers ne sont pas exploités au maximum.

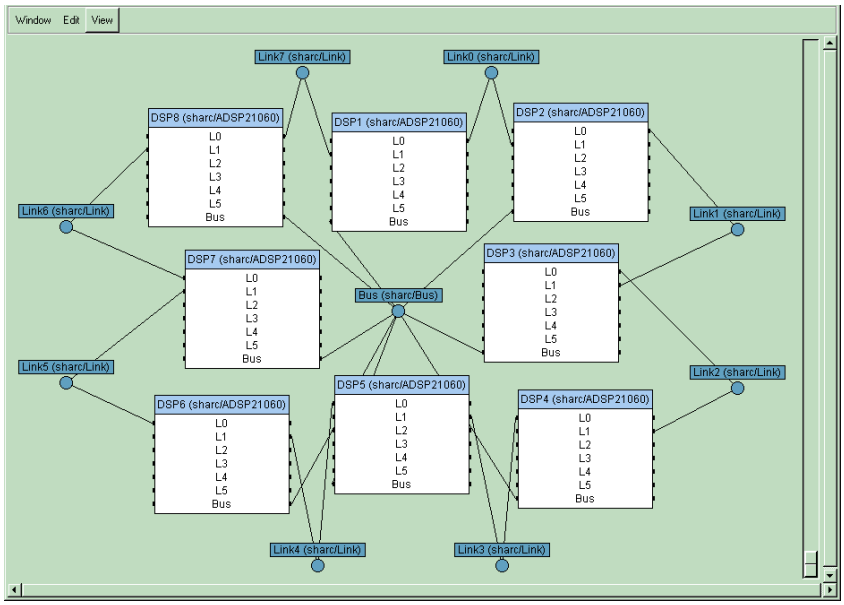


Figure 68 : Architecture à 8 processeurs, avec topologie point à point en anneau +bus.

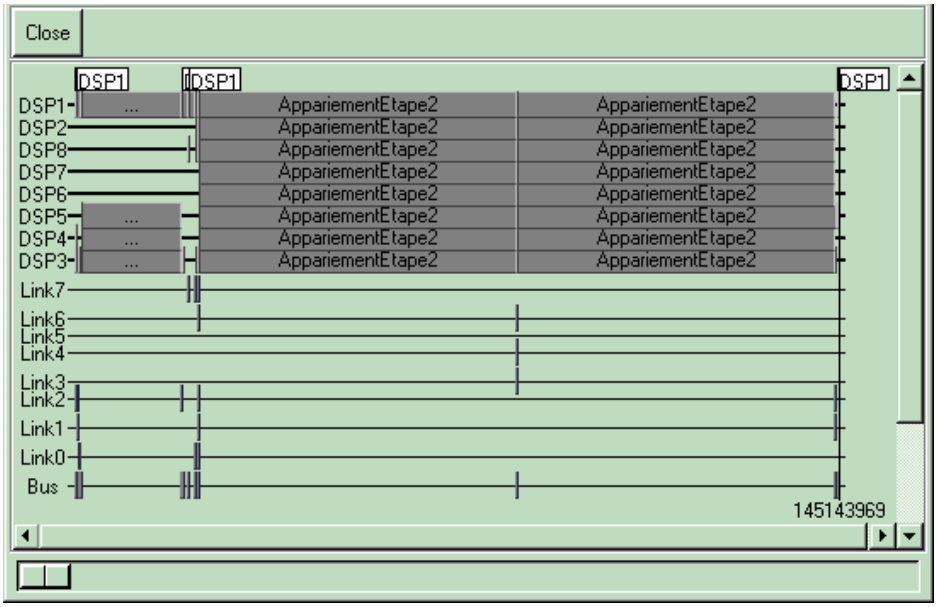


Figure 69 : Distribution/ordonnancement de l'algorithme sur l'architecture à 8 processeurs, avec topologie point à point en anneau +bus.

La durée totale d'exécution de notre algorithme sur une architecture à huit processeurs avec une topologie point à point en anneau avec un bus (Fig. 68) est de 145.143.969 unités de temps (Fig. 69). Nous remarquons, par la présence d'espaces dans les différentes distributions/ordonnancements (Fig. 67 et Fig. 69) que les implantations de notre algorithme sur cette architecture ne sont pas optimisées.

Nbre de processeurs	8	
Topologie	Pt à pt, en anneau	Anneau + Bus
Durée (unités de tps)	145173070	145143969
Gain (en %)	86,222	86,224

5.3.2 Analyse des résultats de l'exploration des différentes implantations

Sur les figures 70, 71 et 72 nous avons représenté les résultats trouvés au §5.3.1 sous forme de graphique. Ces graphiques présentent, pour chacune des topologies, le gain apporté en fonction du nombre de processeurs

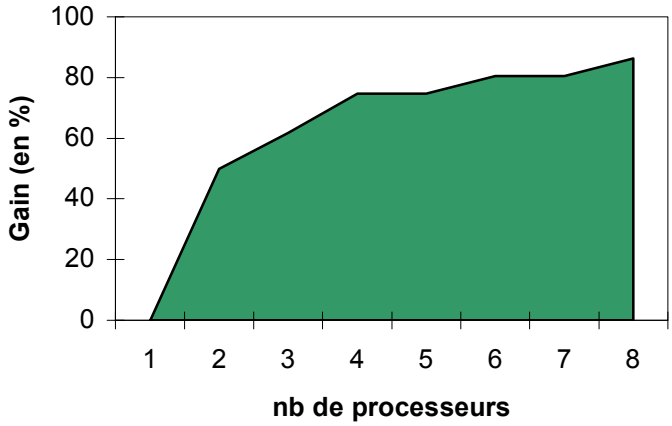


Figure 70 : Topologie point à point en anneau.

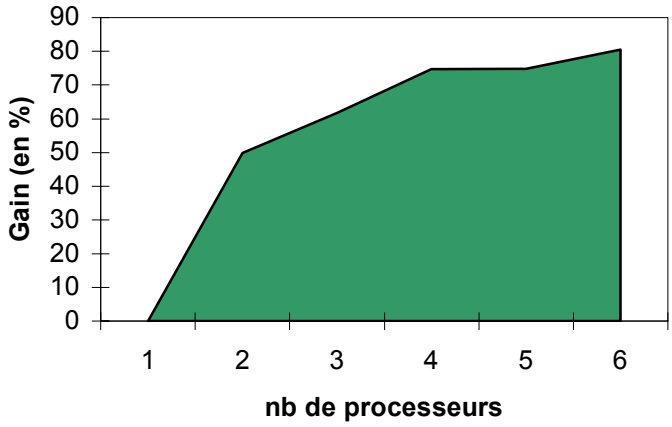


Figure 71 : Topologie point à point complètement connecté.

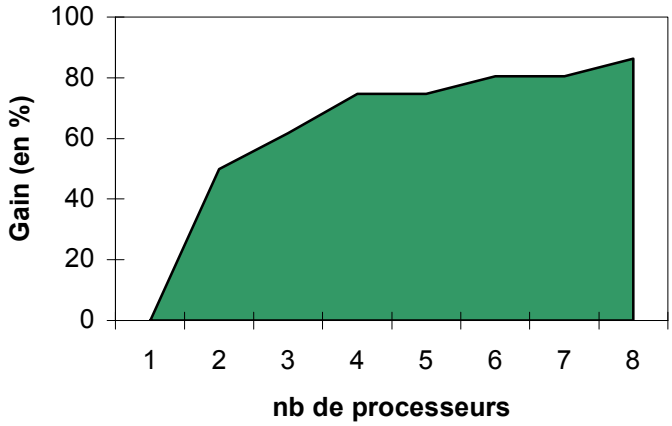


Figure 72 : Topologie point à point en anneau, avec bus.

Nous remarquons que la courbe de gain en fonction du nombre de processeurs est quasiment identique quel que soit la topologie utilisée. En effet, en étudiant les résultats de l'adéquation des implantations de notre algorithme sur les différentes architectures et en fonction des topologies disponibles, nous ne notons qu'une très légère différence entre le gain apporté par la topologie point à point en anneau et la topologie point à point complètement connecté. La topologie point à point en anneau étant plus facile à mettre en oeuvre que la topologie point à point complètement connecté, c'est celle des deux qui sera retenue. Nous aurions noté plus de gain rapporté par la topologie point à point en anneau avec un bus, si nous avions pu caractériser les durées de communication du bus avec ses véritables valeurs. C'est pour cela que la topologie point à point en anneau avec un bus est la topologie que l'on retiendra.

En observant les différentes distribution/ordonnancement obtenues, ainsi que les trois graphiques précédents, on déduit qu'à partir de quatre processeurs, le gain n'évolue que très légèrement et reste autour des 80%. Nous avons relevé précédemment, que les implantations optimisées de notre algorithme s'effectuaient sur l'architecture à quatre processeurs. En effet, les distributions/ordonnancements (Fig. 45, Fig. 47 et Fig. 49) n'ont que quelques faibles espaces, indiquant une bonne exploitation de la puissance de calcul des quatre processeurs. Pour les autres architectures, la présence d'espaces indique une faible exploitation de la puissance de calcul des processeurs.

Pourtant, nous pouvons trouver une implantation de notre algorithme sur une architecture qui est proche d'être optimisée. Sur les figures 67 et 69 les espaces interviennent au début des ordonnancements de quatre des huit processeurs. Nous apercevons sur ces figures, tout au début de l'ordonnancement, quatre blocs nommés « **…** » correspondant aux répétitions de l'opération `CorrelationLigneP`. Si nous avions choisi de spécifier cette opération avec un nombre de répétitions de huit, l'implantation de notre algorithme sur l'architecture à huit processeurs aurait été optimisée. Il existe un lien entre le choix d'un nombre de répétition pour une opération et le résultat de l'adéquation que l'on obtient. En effet, sur les figures 63 et 65 il y a énormément d'espaces dans les ordonnancements des processeurs et cela est dû au fait que nous avons seize répétitions de l'opération `AppariementEtape2` à distribuer sur sept processeurs. Le fait que le nombre de répétitions de l'opération ne soit pas un multiple du nombre de processeurs, cela entraîne que l'implantation n'est pas optimisée. Pour obtenir une implantation optimisée de notre algorithme, il faut que le nombre de répétitions soit un multiple du nombre de processeurs

Nous obtenons une implantation optimisée de notre algorithme sur une architecture à quatre processeurs (car nous avons fixé les nombres de répétitions à quatre) avec une topologie point à point en anneau avec bus.

6 Conclusion

Après une présentation des traitements à réaliser pour déterminer une translation entre deux images, nous avons expliqué l'approche flot de données pour spécifier notre algorithme. A partir de l'algorithme spécifié en flots de données (utilisable avec SynDEx v6), nous avons réalisé une spécification simplifiée (au niveau du parallélisme de données) utilisable par SynDEx v5. Suite à la présentation des principes de SynDEx v5, nous avons décrit toutes les étapes pour effectuer l'implantation complète de notre algorithme en mono-processeur. Une fois le bon fonctionnement de notre implantation en mono-processeur vérifié, nous sommes passés à une implantation en bi-processeur. Après cette phase de simulation, nous avons généré les codes pour les deux processeurs et exécuté ceux-ci sur deux stations de travail. Cette mise en œuvre, correspondant au prototypage, ne respecte pas les contraintes temps réel. Sur l'implantation complète de notre algorithme en bi-processeur nous avons très peu de parallélisme effectif, nous devons donc raffiner la spécification de certaines opérations (dont la durée d'exécution est importante) afin d'exprimer du parallélisme potentiel de données et d'augmenter le parallélisme potentiel d'opérations, en utilisant la nouvelle version v6 du logiciel SynDEx.

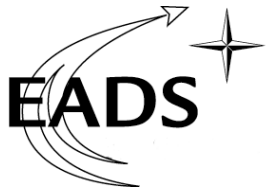
Le passage du logiciel SynDEx v5 à SynDEx v6, apporte de nouvelles fonctionnalités que nous avons présentées. A l'aide de cette nouvelle version du logiciel, nous avons spécifié hiérarchiquement les opérations ayant une durée d'exécution importante, afin de réduire significativement la durée d'exécution totale de notre algorithme. Une fois la description des opérations effectuée, nous avons procédé à l'implantation de l'algorithme complet avec parallélisme de données, et chronométré sur le processeur cible les durées maximales d'exécution des différentes opérations. Nous avons, par la suite, exploré les implantations de notre algorithme sur différentes architectures selon trois type de topologies. Nous avons, à l'issue de cette exploration, retenu une architecture permettant l'implantation optimisée de notre algorithme.

Tout au long de ce rapport, nous avons pu remarquer que le logiciel SynDEx nous permettait de simuler différentes implantations des variantes (avec plus ou moins de parallélisme de données) d'un algorithme sur différentes architectures, afin de rechercher la plus adaptée. Grâce à lui, nous pouvons prédire les performances d'une implantation d'algorithme sur différentes architectures même si nous ne les possédons pas réellement.

La génération de code dans SynDEx v5 à permis de faire du prototypage rapide de notre algorithme sur une architecture réelle à deux stations de travail. Néanmoins cette version ne comportait pas toutes les fonctionnalités nécessaires (spécification hiérarchique, factorisation et conditionnement).

La génération de code n'étant pas encore disponible dans SynDEx v6 qui comprend ces fonctionnalités supplémentaires, la comparaison de simulation avec des résultats d'exécution du même algorithme sur une même architecture matérielle n'a pu être réalisée. Toutefois, grâce aux résultats de la simulation, nous pouvons conclure que le logiciel SynDEx est un outil de prototypage rapide et d'implantation temps réel optimisée pour les applications de traitement d'images.

Annexe 1 : Algorithme de calcul d'une translation entre deux images



Aerospatiale Matra Missiles

Algorithme de calcul d'une translation entre 2 images

E/SCS-2000-759-A du 28 11 2000

Diffusion :
Interne : E/SCS - E/SCS/V – E/SR/P
Externe : CS SI (M. NAKHLE) – INRIA (Y. Sorel) – CEA Leti (F. TERRIER) IRISA (T. GAUTIER) – SITIA (A. ABDALLAH)

Page descriptive seule : E/SC/ID

Aerospatiale Matra Missiles	**N° E/SCS-2000-759-A** **Date 28 11 2000** --- **N° CONTRAT 00 T 230** **CLIENT Ministère de la Recherche**
Classification militaire et industrielle **Document NC Page descriptive NC** **Date de déclassification**	**PROGRAMME ACOTRIS** **MATERIEL** **TECHNIQUE TRAITEMENT D'IMAGE**

TITRE

ALGORITHME DE CALCUL D'UNE TRANSLATION ENTRE 2 IMAGES

RESUME

Au titre du Projet RNTL ACOTRIS, auquel participe : Aérospatiale Matra Missiles (AMM), le CEA-Leti, CS SI, l'INRIA/IRISA et la société SITIA ; AMM se doit :

- **De réaliser une application de référence de Traitement d'Images (TI),**
- **De réaliser une application projet de Traitement d'Images (TI),**
- **De valider la démarche retenue pour le TI.**

L'application de référence sera développée avec SynDEx indépendamment de l'approche ACOTRIS/UML-SIGNAL-SynDEx

L'application projet sera développée avec l'approche ACOTRIS/UML-SIGNAL-SynDEx

Ce document décrit le noyau de l'algorithme de calcul d'une translation entre 2 images qui servira de Spécification Technique de Besoin pour les applications Référence et Projet et donne un synoptique de l'architecture matérielle de la carte sur laquelle sera implémenté l'algorithme, afin que le modèle d'architecture matérielle puisse être réalisé.

AUTEUR (S)
Nom C. SCHLOSSERS
Sigle E/SCS/V
Visa

APPROBATION(S)
Nom N CAMBOU
Sigle E/SCS/V
Visa

Indice	Date	Justifications de l'évolution du document

Nature du document Note technique **Langue FR**	**MOTS CLES** Traitement d'images, algorithme, ACOTRIS	**Nb pages : 31** **Nb figures** **Nb annexes 8**

ARCHIVAGE **Localisation N° interne** **Année destruction** **OBSERVATIONS**	**REFERENCES AUTRES DOCUMENTS**

TABLE DES MATIERES

1. INTRODUCTION

Au titre du Projet RNTL ACOTRIS, auquel participe : Aérospatiale Matra Missiles (AMM), le CEA-Leti, CS SI, l'INRIA/IRISA et la société SITIA ; AMM se doit :

- De réaliser une application de **référence** de Traitement d'Images (TI),
- De réaliser une application **projet** de Traitement d'Images (TI),
- De valider la démarche retenue pour le TI.

L'application de **référence** sera développée avec SynDEx indépendamment de l'approche ACOTRIS/UML-SIGNAL-SynDEx. Cette application permettra également de valider les besoins en modélisation et en expressivité dans les modèles UML.

L'application **projet** sera développée avec l'approche ACOTRIS/UML-SIGNAL-SynDEx. Cette application permettra la validation du processus de co-design (développement conjoint du logiciel et du matériel) retenue pour le Prototypage Rapide (PR) d'applications temps réel de Traitement d'images.

Cette validation sera réalisée en comparant les résultats fournis par l'application de **référence** et l'application obtenue à partir du prototype réalisé dans le **projet** ACOTRIS.

Les résultats seront obtenus en exécutant les simulations de **référence** et **projet** sur tous les scénarios (séquences d'images) de test établis en cohérence avec le domaine d'emploi et les contextes opérationnels des algorithmes définissant l'application.

Nous attacherons un soin particulier à la comparaison de la performance, sur une même architecture matérielle, compte tenu des contraintes imposées par chacune des démarches (**référence** et **projet**) :

- Des algorithmes (précision, robustesse…),
- Des temps de traitement.

Ce document décrit donc le noyau de **l'algorithme de calcul d'une translation entre 2 images** qui servira de Spécification Technique de Besoin pour les applications **Référence** et **Projet**.

Un synoptique de l'architecture matérielle de la carte, sur laquelle sera implémenté l'algorithme, est donnée, afin que le modèle d'architecture matérielle puisse être réalisée. La carte de traitement d'images temps réel, qui sera utilisée, est équipée d'ASICS pour les traitements de bas niveau et de 4 DSP SHARC 21060 pour le haut niveau.

L'algorithme est fondé sur une succession de calculs de translation en repère image entre 2 images successives d'une séquence. Il utilise des points caractéristiques extraits par un des ASIC sur chaque image. L'appariement des points caractéristiques homologues s'effectue par corrélation locale inter image. Une translation globale en x et y est ensuite calculée entre 2 images successives.

On trouvera dans les annexes les fichiers source et d'en-tête du logiciel de calcul de la translation inter image.

2. TRANSLATION INTER-IMAGE

Le calcul de la translation entre 2 images successives d'une scène prise par une caméra mobile se décompose en 3 parties de la façon suivante :

1. Extraction des points caractéristiques de l'image 1. Elle est réalisée par un algorithme de Plessey implémenté dans l'Asic OPNI. L'image et la liste de points sont sauvegardées en mémoire. Il en est de même pour l'image 2.
2. Appariement des points caractéristiques homologues de l'image 1 et de l'image 2. Cet algorithme, gourmand en temps, sera parallélisé.
3. Calcul de la translation Tx, Ty. Une méthode simple calcule les translations (tx,ty) de chaque point apparié. La translation médiane est retenue.

3. APPARIEMENT DE POINTS CARACTERISTIQUES

Ce paragraphe décrit l'algorithme de l'appariement de points caractéristiques par corrélation. Il donne les paramètres d'entrées, de sorties et de traitement.

3.1. ALGORITHME D'APPARIEMENT DE POINTS CARACTERISTIQUES PAR CORRELATION

3.1.1. Pré-estimation de la translation globale entre les images

On commence par réduire les images d'un facteur 2 pour aller plus vite, puis on effectue une corrélation du centre de l'image 1 dans toute l'image 2. Ceci fournit une translation estimée globale.

3.1.2. Appariement des points caractéristiques

Pour tous les points caractéristiques extraits de l'image 1

Pour tous les points caractéristiques extraits de l'image 2

On effectue une première corrélation simple sans déplacement autour de la

position du plot

Si le score OK

Corrélation fine avec estimation de la position précise du Plot, et calcul du score

On garde le point avec le score minimum

Fin points de l'image 2

Calcul de la confiance pour le point avec le score minimum

Vérification si pas de conflit d'unicité

Fin points de l'image 1

3.1.3. Filtrage des appariements incorrects

On calcule le vecteur translation pour chaque point, et on considère qu'un appariement est correct si un pourcentage minimum de vecteurs translation des autres appariements est compatible avec lui.

3.1.4. Paramétrage

Les entrées

image 1 : 256*256, codée sur 8 bits

liste 1 des points caractéristiques extraits

image 2 : 256*256, codée sur 8 bits

liste 2 des points caractéristiques extraits

Les paramètres de traitement

distx :	Distance de recherche des plots en x
disty :	Distance de recherche des plots en y
guess :	Pré-estimation de la translation globale entre les images par corrélation du centre de la première image
fast :	Fait une 1ere corrélation simple
win :	Taille de la fenêtre de corrélation
var :	Précision de localisation des plots
seuil :	Différence de niveau de gris maximum entre plots
conf :	Confiance minimum sur les appariements, la confiance est le pourcentage de différence avec l'appariement de score le plus proche
varmvt :	Variation des vecteurs déplacement tolérée
check :	Pourcentage de vecteurs translation devant être cohérents

Les sorties

Translation sub-pixellique Tx, Ty

3.1.5. Fichiers

Fichiers sources :

match_plots.c	:	fichier principal	Annexe 1-1
matchplots.c	:	appariement	Annexe 1-2
correl.c	:	corrélation	Annexe 1-3
checkmatches.c	:	contrôle des appariements	Annexe 1-4
Data.c	:	variables en mémoire partagée	Annexe 1-5
tools.c	:	outils	Annexe 1-6

Fichiers en-tête :

match_plots.h	:	Annexe 1-7
type_a.h	:	Annexe 1-8

3.2. SYNOPTIQUE DE L'ARCHITECTURE MATERIELLE

ANNEXE 1-1

```
/*************************************************************************
*
*     MODULE:     match_plots.c                        Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: s'execute sur un processeur PENTIUM
*                       outils de communication
*
*     CONTENU:     main,  ,  ,
*
*
*     HISTORIQUE:       creation
*                            20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************/
/* fichiers directives systeme */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
/* #include <time.h> */


#include "M_plots.h"
#include "type_a.h"

extern      float m1[][2], scores[];
extern      float m2[][2], confidence[];

#define     PRINT
/* #define  ORIENTATION */

/* jpl 24 10 2000
unsigned char     im1[WIDTH*HEIGHT], im2[WIDTH*HEIGHT];
U2BT  im1reduc[WIDTH_2*HEIGHT_2], im2reduc[WIDTH_2*HEIGHT_2];
*/
int         plots1[MAX_PLOT][2], origrad1[MAX_PLOT], numplots1 = 0;
int         plots2[MAX_PLOT][2], origrad2[MAX_PLOT], numplots2 = 0;


extern      unsigned char     im1[WIDTH*HEIGHT];
extern      unsigned char     im2[WIDTH*HEIGHT];
extern      U2BT  im1reduc[WIDTH_2*HEIGHT_2];
extern      U2BT  im2reduc[WIDTH_2*HEIGHT_2];
/*
extern      int         plots1[MAX_PLOT][2];
extern      int         origrad1[MAX_PLOT];
extern      int         numplots1;
extern      int         plots2[MAX_PLOT][2];
extern      int         origrad2[MAX_PLOT];
extern      int         numplots2;
*/


int center[2];
float res[2];
```

```
/* for benchmark routines */
int count_start();
int count_end(int);

int   time_temp,cycle_count;

void main (void)
{
  char            nomIm1[64], nomIm2[64], nomPlots1[64], nomPlots2[64];
  char            nomMatches[64];

  int       typeIm, distx, disty = -1, guess, fast, win, var, varmvt;
  float           seuil, conf;
  float           check;
  int       width, height;
  int       genre;

  int       (*pred)[2] = NULL;
  int       nummatches, numcorrect = 0;
  int       globaltrans[2] = {0, 0};

  int i, j, k;


    /* Variables for  time measurement */
      /* clock_t  tStart, tEnd; */

      /* initialisations */
      strcpy(nomIm1,"Image/image3");
      strcpy(nomIm2,"Image/image4");

      strcpy(nomMatches,"match1.dzv");

      typeIm      = 3;
      guess = 0;
      fast  = 1;
      distx = 30;
      disty = 30;
      var         = 1;
      varmvt      = 10;
      check = 10.0;
      win         = 9;
      seuil = 20.0;
      conf  = 15.0;
      genre = 1;
      width = WIDTH;
      height      = HEIGHT;

    for (j = 0; j < height; j++) {
      for (i = 0; i < width; i++) {
        im1[j*width+i]=0;
        im2[j*width+i]=0;
      }
    }
  /*
   * lecture des images d'entree
   */

      read_file(nomIm1,width,height,im1);
    for (j = 0; j < height; j++) {
```

```
      for (i = 0; i < width; i++) {
            im1[j*width+i] &= 0x0ff;
            /*printf("0x%x\t",im1[j*width+i]);*/
      }
    }
      read_file(nomIm2,width,height,im2);
    for (j = 0; j < height; j++) {
      for (i = 0; i < width; i++) {
            im2[j*width+i] &= 0x0ff;
            /*printf("0x%x\t",im2[j*width+i]);*/
      }
    }

  /*
   * lecture des plots
   */

      read_plots_SansOri("Image/PlesseySimu3.txt",plots1,&numplots1);
#ifdef      PRINT
  printf("match_plots.c: nbre plots 1: %d\n",numplots1);
#endif


  read_plots_SansOri("Image/PlesseySimu4.txt",plots2,&numplots2);
#ifdef      PRINT
  printf("match_plots.c: nbre plots 2: %d\n",numplots2);
#endif


  /*
   * Pre-estimation de la translation globale entre les images.
   * On commence par reduire les images d'un facteur 2 pour aller plus vite,
   * puis on effectue une correlation du centre de l'image 1 dans toute
   * l'image 2.
   */

    /* time measurement */
    /* tStart = clock(); */
   time_temp = count_start();                    /* Benchmark routine start */

  if (guess) {


#ifdef      PRINT
      printf("processing pre-estimation\n");
      printf("res: 0x%x\n",res);
#endif

    k = (width >> 1) * (height >> 1);
      for (i = 0; i < k; i++) {
            im1reduc[i] = 0;
            im2reduc[i] = 0;
      }

    for (j = 0; j < height; j++) {
      for (i = 0; i < width; i++) {
        k = (j >> 1)*(width >> 1) + (i >> 1);
        im1reduc[k] += im1[j*width+i];
        im2reduc[k] += im2[j*width+i];
      }
    }
```

```
    center[0] =  width >> 2;
    center[1] = height >> 2;

    if (correl((char *)im1reduc, (char *)im2reduc, U2BTIMAG,
        width >> 1, height >> 1, center, center, width >> 4, width >> 2,
        4*seuil, res)) {

      globaltrans[0] = (int)(res[0] - center[0]) * 2;
      globaltrans[1] = (int)(res[1] - center[1]) * 2;
    }
#ifdef      PRINT
      printf("center[0] x: %d\n",center[0]);
      printf("center[1] y: %d\n",center[1]);
      printf("res[0] x: %f\n",res[0]);
      printf("res[1] y: %f\n",res[1]);
      printf("Translation globale en x: %d\n",globaltrans[0]);
      printf("Translation globale en y: %d\n",globaltrans[1]);
#endif
  }

/*globaltrans[0] = -36;
  globaltrans[1] = 0;*/


  /*
   * appariement des plots
   */

#ifdef      ORIENTATION
#ifdef      PRINT
      printf("Processing with orientation !!!\n");
#endif


  nummatches = matchplots(im1, im2, genre, width, height,
    plots1, origrad1, pred, plots2, origrad2, numplots1, numplots2,
globaltrans,
    distx, disty, win, var, seuil, conf, fast);
#else
#ifdef      PRINT
      printf("Processing without orientation !!!\n");
#endif

  nummatches = matchplots(im1, im2, genre, width, height,
    plots1, NULL, pred, plots2, NULL, numplots1, numplots2, globaltrans,
    distx, disty, win, var, seuil, conf, fast);
#endif

  /*
   * filtrage des appariements incorrects
   */

  if (check > 0) {
    numcorrect = checkmatches(m1, m2, scores, nummatches, varmvt, check);
  }

  /* time measurement */
  /* tEnd = clock(); */
  cycle_count = count_end(time_temp);          /* Benchmark routine end   */
```

```
  printf(" Correlation Points caracteristiques:  %d cycles, %d micros \n",
cycle_count, cycle_count/1000*25);

  /* printf("Processing time = %2.4f seconds\n",(float)(tEnd - tStart) /
CLOCKS_PER_SEC); */

  printf("nummatches: %d, numcorrect: %d\n",nummatches,numcorrect);


  /*
   * ecriture des appariements
   */

  /**/
  write_matches("Plessey3.asc", m1, m2, nummatches, numcorrect,
                scores, confidence, width, height);
  /**/

  exit(0);
}
```

ANNEXE 1-2

```
/*************************************************************************
*
*
*     MODULE:     matchplots.c                          Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: s'execute sur un processeur PENTIUM
*                        outils de communication
*
*     CONTENU:     matchplots(),  ,  ,
*
*
*     HISTORIQUE:       creation
*                             20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#include "M_plots.h"
#include "type_a.h"

int   matches[MAX_PLOT];

float m1[MAX_PLOT][2], scores[MAX_PLOT];
float m2[MAX_PLOT][2], confidence[MAX_PLOT];

int matchplots(unsigned char *im_prev, unsigned char *im, int genre,
  int dimx, int dimy,
  int (*plots_prev)[2], int *origrad_prev, int (*pred_prev)[2],
  int (*plots)[2], int *origrad, int numplots_prev, int numplots,
  int globaltrans[2], int distx, int disty, int win, int var,
  float seuil_ng, float seuil_conf, int fast)
{
  int       halfwin, i, j, conflict, diff;
  float     score, minscore, secondminscore;
  int       nummatches, numalloc;
  int       plot_cur[2], dx, dy;
  float     res[2], conf;

  numalloc = MAX_PLOT;


  halfwin = (win - 1)/2;
  nummatches = 0;
  for (i = 0; i < numplots_prev; i++) {

    /* on ne traite pas les plots trop proches des bords de l'image */
    if (plots_prev[i][0] < halfwin || plots_prev[i][0] >= dimx - halfwin)
continue;
    if (plots_prev[i][1] < halfwin || plots_prev[i][1] >= dimy - halfwin)
continue;

    minscore = secondminscore = -1;
    for (j = 0; j < numplots || (numplots < 0 && j == 0); j++) {

      if (numplots >= 0) {
        plot_cur[0] = plots[j][0];
        plot_cur[1] = plots[j][1];
```

```
      } else {
        if (pred_prev) {
          plot_cur[0] = pred_prev[i][0];
          plot_cur[1] = pred_prev[i][1];
        } else {
          plot_cur[0] = plots_prev[i][0];
          plot_cur[1] = plots_prev[i][1];
        }
      }

      /* on ne traite pas les plots trop proches des bords de l'image */
      if (plot_cur[0] < halfwin || plot_cur[0] >= dimx - halfwin) continue;
      if (plot_cur[1] < halfwin || plot_cur[1] >= dimy - halfwin) continue;

      if (numplots >= 0) {

        /* Teste l'orientation des plots */
        if (origrad) {
          diff = abs(origrad_prev[i] - origrad[j]);
          if (diff > 8) diff = 16 - diff;
          if (diff > 1) continue;
        }

        /* utilise la prediction si elle est disponible */
        if (pred_prev) {
          dx = plot_cur[0] - pred_prev[i][0];
          dy = plot_cur[1] - pred_prev[i][1];
        } else {
          dx = plot_cur[0] - plots_prev[i][0] - globaltrans[0];
          dy = plot_cur[1] - plots_prev[i][1] - globaltrans[1];
        }

        if (abs(dx) > distx || abs(dy) > (disty >= 0 ? disty : distx))
continue;
      }

      /*
       * On effectue une premiere correlation simple sans deplacement autour
       * de la position du plot, avec une tolerance 1.5 fois superieure.
       */

      if (fast) {

        register int    ii, jj;
        int size, nextline;
        register unsigned char     *p1, *p2;

        switch (genre) {
          case U1BTIMAG : size = 1; break;
          case U2BTIMAG : size = 2; break;
          default : fprintf(stderr, "Erreur : images U1BT ou U2BT
seulement\n");
                    exit(1);
        }

        nextline = (dimx - (2*halfwin+1))*size;
        p1 = im_prev + (plots_prev[i][0] - halfwin + dimx * (plots_prev[i][1]
- halfwin)) * size;
        p2 = im      + (plot_cur[0]      - halfwin + dimx * (plot_cur[1]
- halfwin)) * size;
        score = 0;
```

```
        for (jj = -halfwin; jj <= halfwin; jj++, p1 += nextline, p2 +=
nextline) {
          for (ii = -halfwin; ii <= halfwin; ii++, p1 += size, p2 += size) {
            if (genre == U1BTIMAG) score += abs(*(U1BT *)p2 - *(U1BT *)p1);
            else           score += abs(*(U2BT *)p2 - *(U2BT *)p1);
          }
        }
        if (score > 1.5*seuil_ng * win*win) continue;
      }

      /*
       * Correlation avec estimation de la position precise du plot
       */

      score = (float)correl(im_prev, im, genre, dimx, dimy, plots_prev[i],
plot_cur,
        halfwin, var, seuil_ng, res);

      if (score < 0) continue;

      if (minscore < 0 || score < minscore) {

        secondminscore = minscore;
        minscore = score;
        matches[nummatches] = j;
        m1[nummatches][0] = plots_prev[i][0];
        m1[nummatches][1] = plots_prev[i][1];
        m2[nummatches][0] = res[0];
        m2[nummatches][1] = res[1];

      } else if (score < secondminscore) {
             secondminscore = score;
        }
    } /* fin du for: j < numplots || (numplots < 0 && j == 0) */

    if (minscore < 0) continue;

    /* Calcul de confiance */

    conf = (secondminscore < 0)? 1 : (secondminscore -
minscore)/secondminscore;
    conf *= 100;

    if (conf < seuil_conf) continue;

    /* Normalisation du score de correlation */
    minscore /= win*win;

    /* Verifie qu'il n'y a pas de conflit d'unicite */
    for (j = 0, conflict = 0; numplots >= 0 && j < nummatches; j++) {
      if (matches[j] == matches[nummatches]) {
        /* si le nouveau score est meilleur, on remplace l'ancien appariement
           par le nouveau, sinon on rejette le nouveau */
        if (scores[j] > minscore) {
/* printf("Anc: x:%f, y:%f, Nouv: x:%f,
y:%f\n",m1[j][0],m1[j][1],m1[nummatches][0],m1[nummatches][1]); */
          m1[j][0] = m1[nummatches][0];
          m1[j][1] = m1[nummatches][1];
          m2[j][0] = m2[nummatches][0];
          m2[j][1] = m2[nummatches][1];
          scores[j] = minscore;
```

```
        }
        conflict = 1;
        break;
      }
    }
    if (conflict) continue;

    scores[nummatches] = minscore;
    confidence[nummatches] = conf;
    nummatches++;

    if (nummatches >= numalloc) {

            return(nummatches);
    }
  } /* fin du for: i < numplots_prev */


  return(nummatches);
}
```

ANNEXE 1-3

```
/*************************************************************************
*
*
*     MODULE:     correl.c                      Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: s'execute sur un processeur PENTIUM
*                       outils de communication
*
*     CONTENU:     correl(),  ,  ,
*
*
*     HISTORIQUE:       creation
*                             20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#include "M_plots.h"
#include "type_a.h"

/*-----------------------------------------------------------------------*/
/* Correlation avec interpolation subpixellique.                         */
/* Retourne le score de correlation ou -1 si echec.                      */
/*                                                    */
/*                                                    */
/* L'idee utilisee pour l'interpolation subpixellique est issue de :     */
/* Zhang, Z., "A Stereovision System for a Planetary Rover:              */
/* Calibration, Correlation, Registration, and Fusion", in Proc. IEEE    */
/* Workshop on Planetary Rover Technology and Systems, Apr. 23, 1996,    */
/* Minneapolis, Minnesota, USA.                                          */
/* ftp://krakatoa.inria.fr/pub/html/Papers/zhang:96b.ps.gz               */
/*                                                    */
/* Parametres :                                            */
/*                                                    */
/* im1, im2     : pointeurs sur les deux images codees sur 1 ou 2*/
/*                octets                                         */
/* genre        : codage des images (U1BTIMAG ou U2BTIMAG)       */
/* dimx, dimy         : taille des images                        */
/* pos1[0], pos1[1] : position du point a correler dans l'image 1      */
/* pos2[0], pos2[1] : centre de la fenetre de recherche dans l'image 2 */
/* halfwin      : demi largeur du motif de correlation           */
/* var                : demi largeur zone de recherche dans l'image 2  */
/* seuil_ng     : seuil sur le cout de correlation               */
/* res[0], res[1]   : resultat : position subpixellique du maximum de  */
/*                correlation dans l'image 2                     */
/*-----------------------------------------------------------------------*/

/*long int  lscores[MAXSCORESIZE];*/


float correl(unsigned char *im1, unsigned char *im2, int genre,
  int dimx, int dimy, int pos1[2], int pos2[2], int halfwin, int var,
  float seuil_ng, float res[2])
{
  long int  lscores[MAXSCORESIZE];
  register int               i, j;
  register unsigned char     *p1, *p2, *p;
```

```
long int                    imin = 0, jmin = 0;
long int                    di, dj, di_min, di_max, dj_min, dj_max;
int                   win, nextline, size;
static int            scoresize = MAXSCORESIZE;
long int              *pscore, minscore;
long int              z1, z2, z3, z4, z5, a2, b2, c2, d2;
float                       dx, dy;
float                       ret;


switch (genre) {
  case U1BTIMAG : size = 1; break;
  case U2BTIMAG : size = 2; break;
  default : fprintf(stderr, "Erreur : images U1BT ou U2BT seulement\n");
            exit(1);
}


/*
 * ajuste la fenetre de recherche pour ne pas sortir de l'image
 */

/*
 * on augmente de 1 la zone de recherche pour avoir un voisinage
 * d'interpolation 3x3 meme sur les bords
 */

var += 1;

if (pos2[0] - halfwin - var >= 0) {
  di_min = -var;
} else {
  di_min = halfwin - pos2[0];
}
if (pos2[0] + halfwin + var < dimx) {
  di_max = var;
} else {
  di_max = dimx-1 - halfwin - pos2[0];
}
if (pos2[1] - halfwin - var >= 0) {
  dj_min = -var;
} else {
  dj_min = halfwin - pos2[1];
}
if (pos2[1] + halfwin + var < dimy) {
  dj_max = var;
} else {
  dj_max = dimy-1 - halfwin - pos2[1];
}

if (di_max-di_min+1 < 3 || dj_max-dj_min+1 < 3) {
  return(-1);
}


/*
 * Alloue un tableau pour stocker les valeurs de correlation a interpoler.
 * Optimisation : on conserve l'espace alloue aux appels precedants.
 */

if ((dj_max-dj_min+1)*(di_max-di_min+1) > scoresize) {
```

```
      printf("Tableau des scores excedent la taille alloue: %d\n",
            (dj_max-dj_min+1)*(di_max-di_min+1));
      return(-1);
  }



  win = 2*halfwin+1;
  nextline = (dimx - win)*size;

  /* parcours de la zone de recherche */
  p = im1 + (pos1[0] - halfwin + dimx * (pos1[1] - halfwin)) * size;
  for (dj = dj_min, pscore = lscores; dj <= dj_max; dj++) {
    for (di = di_min; di <= di_max; di++, pscore++) {

      p1 = p;
      p2 = im2 + (pos2[0] + di - halfwin + dimx * (pos2[1] + dj - halfwin)) *
size;
      *pscore = 0;

      /* parcours de la fenetre de correlation */
      /* les calculs de pointeurs dans les images sont tres optimises */

      if (genre == U1BTIMAG) {

        for (j = -halfwin; j <= halfwin; j++, p1 += nextline, p2 += nextline)
{
          for (i = -halfwin; i <= halfwin; i++, p1 += size, p2 += size) {
            *pscore += abs(*(U1BT *)p2 - *(U1BT *)p1);
          }
        }
      } else {

        for (j = -halfwin; j <= halfwin; j++, p1 += nextline, p2 += nextline)
{
          for (i = -halfwin; i <= halfwin; i++, p1 += size, p2 += size) {
            *pscore += abs(*(U2BT *)p2 - *(U2BT *)p1);
          }
        }
      }
    }
  }


  /*
   * Recherche du pic de correlation discret
   */

  minscore = -1;
  for (j = dj_min, pscore = lscores; j <= dj_max; j++) {
    for (i = di_min; i <= di_max; i++, pscore++) {
      if (minscore < 0 || *pscore < minscore) {
        minscore = *pscore;
        imin = i;
        jmin = j;
      }
    }
  }

  /*
   * Si le pic de correlation est sur un bord, ce n'est probablement pas
```

```
 * un minimum local, donc on invalide l'appariement.
 * Si la valeur de correlation est trop elevee, invalide l'appariement.
 */

if (minscore < 0 ||
    imin == di_min || imin == di_max ||
    jmin == dj_min || jmin == dj_max ||
    minscore > seuil_ng * win*win) {
  return(-1);
}

/*
 * Resultat de la correlation discrete.
 */

res[0] = pos2[0] + imin;
res[1] = pos2[1] + jmin;


/*
 * Determination de la position subpixellique du pic de correlation.
 * On approxime les valeurs de correlation par une fonction quadratique
 * sur un voisinage en 4-connexite :
 *
 *    z1
 * z2 z3 z4
 *    z5
 *
 *  z = a x^2 + b y^2 + c x + d y + f
 *
 * d'ou :
 *
 *  a = 1/2 (z2 + z4 - 2 z3)
 *  b = 1/2 (z1 + z5 - 2 z3)
 *  c = 1/2 (z4 - z2)
 *  d = 1/2 (z5 - z1)
 *  f = z3
 *
 * Position du minimum :
 *
 *  x = - c / 2a
 *  y = - d / 2b
 */

pscore = lscores + imin-di_min + (jmin-dj_min-1)*(di_max-di_min+1);
z1 = *pscore;
pscore += di_max-di_min+1-1;
z2 = *pscore;
pscore ++;
z3 = *pscore;
pscore ++;
z4 = *pscore;
pscore += di_max-di_min+1-1;
z5 = *pscore;

a2 = z2 + z4 - 2 * z3;
b2 = z1 + z5 - 2 * z3;
c2 = z4 - z2;
d2 = z5 - z1;

if (a2 == 0) {
```

```
    dx = 0;
  } else {
    dx = - c2 / (float)(2 * a2);
  }

  if (b2 == 0) {
    dy = 0;
  } else {
    dy = - d2 / (float)(2 * b2);
  }

  res[0] += dx;
  res[1] += dy;

  ret = (float)minscore;

  return(ret);
}
```

ANNEXE 1-4

```
/*************************************************************************
*
*
*     MODULE:     checkmatches.c                        Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: s'execute sur un processeur PENTIUM
*                         outils de communication
*
*     CONTENU:     checkmatches(),  ,  ,
*
*
*     HISTORIQUE:       creation
*                              20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#include "M_plots.h"


int checkmatches (float (*m1)[2], float (*m2)[2], float *scores,
  int nummatches, int varmvt, float check)
{
  float     dx1, dy1, dx2, dy2;
  int       numvois, minvois, numcorrect;
  /*int           (*match1)[2], (*match2)[2];*/
  int       i, j;

  /*
   * On considere qu'un appariement est correct si au moins check % des
autres
   * appariements sont compatibles avec lui.
   */

  minvois = (int)(nummatches * check / 100.);
  if (minvois < 1) minvois = 1;

  numcorrect = nummatches;
  for (i = 0; i < nummatches; i++) {

    dx1 = m2[i][0] - m1[i][0];
    dy1 = m2[i][1] - m1[i][1];

    numvois = 0;
    for (j = 0; j < nummatches; j++) {
      /* if (match2 == match1) continue; */

      dx2 = m2[j][0] - m1[j][0];
      dy2 = m2[j][1] - m1[j][1];

      if ((int)fabs(dx2 - dx1) <= varmvt && (int)fabs(dy2 - dy1) <= varmvt) {
        numvois++;
        if (numvois >= minvois) break;
      }
    }

    if (numvois < minvois) {
```

```
      scores[i] = -1;
      numcorrect--;
    }
  }

  return(numcorrect);
}
```

ANNEXE 1-5

```
/*************************************************************************
*
*
*     MODULE:     Data.c                          Version     1.0
*
*     PROJET: Carte TI embarquée - copyright Aerospatiale 2000
*
*     DESCRIPTION: Appariement Par le Contexte + Poursuite
*
*
*     CONTENU:
*
*
*     HISTORIQUE:       creation
*                             26/05/00    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#include "M_plots.h"
#include "type_a.h"

/*long int  lscores[MAXSCORESIZE];*/

unsigned char     im1[WIDTH*HEIGHT];
unsigned char     im2[WIDTH*HEIGHT];
U2BT  im1reduc[WIDTH_2*HEIGHT_2], im2reduc[WIDTH_2*HEIGHT_2];

/*
int         plots1[MAX_PLOT][2], origrad1[MAX_PLOT], numplots1 = 0;
int         plots2[MAX_PLOT][2], origrad2[MAX_PLOT], numplots2 = 0;
*/
```

ANNEXE 1-6

```
/*************************************************************************
*
*
*     MODULE:     tools.c                        Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: s'execute sur un processeur PENTIUM
*                       outils de communication
*
*     CONTENU:    read_file_Surf, write_file_Surf, read_file, write_file
*                       read_plots, write_plots, write_matches
*
*
*     HISTORIQUE:       creation
*                             05/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#include <stdio.h>
#include <stdlib.h>



/*********************************************************************
*
* read_file() - Read the image on file and load it on buffer
*
*********************************************************************/
void    read_file(char *pcFile,int iTailleX,int iTailleY,char *pcBuff)
{
    FILE *pfHandlefile;

    int taille_file=iTailleX*iTailleY;

    /* ouverture du fichier */
    pfHandlefile=fopen(pcFile,"rb");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
    }

    /* lecture du fichier */
     if (fread8(pcBuff, 1, (int) (taille_file), pfHandlefile)==-1){
         /*perror("pb read:");*/
     }
     fclose(pfHandlefile);

}

/*********************************************************************
*
* write_file() - Write the image on file
*
*********************************************************************/
void    write_file(char *pcFile,int iTailleX,int iTailleY,char *pcBuff)
{
```

```
    FILE *pfHandlefile;

    int taille_file=iTailleX*iTailleY;

    /* ouverture du fichier */
    pfHandlefile=fopen(pcFile,"wb");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
    }

    /* ecriture de la memoire dans le fichier */
    if (fwrite(pcBuff, 1, (int) (taille_file), pfHandlefile)==-1){
        /*perror("pb write\n");*/
    }
    fclose(pfHandlefile);

}


/*********************************************************************
*
* read_plots() - Read the plots on file and load it on buffer
*
*********************************************************************/
void  read_plots(char *pcFile,int (*piPlots)[2],int *piOri,int *iNbre)
{
    FILE *pfHandlefile;
      int   iBoucle;


      /* ouverture du fichier */
      pfHandlefile=fopen(pcFile,"r");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
      }

      /* Nombre de plots dans l'image */
      fscanf(pfHandlefile,"%d\n",iNbre);


    /* ecriture de la memoire dans le fichier */
      for( iBoucle = 0; iBoucle < *iNbre; iBoucle++){
        fscanf(pfHandlefile,"%d %d
%d\n",&piPlots[iBoucle][0],&piPlots[iBoucle][1],&piOri[iBoucle]);
      }

     fclose(pfHandlefile);

}

/*********************************************************************
*
* read_plots_SansOri() - Read the plots on file and load it on buffer
*
*********************************************************************/
void  read_plots_SansOri(char *pcFile,int (*piPlots)[2],int *iNbre)
{
```

```
    FILE *pfHandlefile;
      int   iBoucle;


      /* ouverture du fichier */
      pfHandlefile=fopen(pcFile,"r");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
      }

      /* Nombre de plots dans l'image */
      fscanf(pfHandlefile,"%d\n",iNbre);


    /* ecriture de la memoire dans le fichier */
      for( iBoucle = 0; iBoucle < *iNbre; iBoucle++){
        fscanf(pfHandlefile,"%d
%d\n",&piPlots[iBoucle][0],&piPlots[iBoucle][1]);
      }

     fclose(pfHandlefile);

}

/*********************************************************************
*
* write_plots() - Write the plots on file
*
*********************************************************************/
void  write_plots(char *pcFile,int (*piPlots)[2],int *piOri,int *iNbre)
{
    FILE *pfHandlefile;
      int   iBoucle;

      /* ouverture du fichier */
      pfHandlefile=fopen(pcFile,"w");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
      }

      /* Nombre de plots dans l'image */
      fprintf(pfHandlefile,"%d\n",*iNbre);

    /* ecriture de la memoire dans le fichier */
      for( iBoucle = 0; iBoucle < *iNbre; iBoucle++){
            fprintf(pfHandlefile,"%d %d
%d\n",piPlots[iBoucle][0],piPlots[iBoucle][1],piOri[iBoucle]);
      }

    fclose(pfHandlefile);

}

/*********************************************************************
*
* write_matches() - Write the matches on file
*
```

```
**********************************************************************/
void write_matches(pcFile, m1, m2, nummatches, numcorrect, scores,
confidence, width, height)
char  *pcFile;
float (*m1)[2];
float (*m2)[2];
int         nummatches;
int         numcorrect;
float *scores;
float *confidence;
int         width;
int         height;
{
    FILE *pfHandlefile;
      int   iBoucle;

      /* ouverture du fichier */
      pfHandlefile=fopen(pcFile,"w");
     if (pfHandlefile==NULL) {
        printf("le fichier %s n'existe pas\n",pcFile);
        /*perror("pb open:");*/
        exit;
      }

      /* Taille de l'image */
      fprintf(pfHandlefile,"%d %d\n",width,height);

      /* Nombre de matches dans l'image */
      fprintf(pfHandlefile,"%d\n",numcorrect);

    /* ecriture de la memoire dans le fichier */
      for( iBoucle = 0; iBoucle < nummatches; iBoucle++){
            if( scores[iBoucle] < 0 )continue;
            fprintf(pfHandlefile,"%f %f %f %f %f
%f\n",m1[iBoucle][0],m1[iBoucle][1],m2[iBoucle][0],m2[iBoucle][1],scores[iBou
cle],confidence[iBoucle]);
      }

    fclose(pfHandlefile);

}
/*********************************************************************/
/*               Procedure copyU1BT (U1BT en U1BT)                   */
/*********************************************************************/
void CopyIMG(unsigned char *src,unsigned char *dest, int nlig, int ncol)
{
 register int  nbpix,i;
 unsigned char   *pts;
 unsigned char   *ptd;

 nbpix = ncol * nlig;

 pts = (unsigned char *)src;
 ptd = (unsigned char *)dest;

 for(i=0;i<nbpix;i++,pts++,ptd++)
    *ptd = *pts;

}
```

ANNEXE 1-7

```
/*************************************************************************
*
*
*     MODULE:     match_plots.h                        Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: fichier de definition
*
*     CONTENU:     ,  ,  ,
*
*
*     HISTORIQUE:       creation
*                             20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#ifndef     _MATCH_PLOTS_H

#include <stdlib.h>
#include <stdio.h>
/* #include <malloc.h> */
#include <math.h>

#define     MAX_PLOT    500
#define     MAXSCORESIZE      10000
#define     HEIGHT            256
#define     WIDTH       256
#define     HEIGHT_2    HEIGHT/2
#define     WIDTH_2           WIDTH/2

extern int matchplots(unsigned char *im_prev, unsigned char *im, int genre,
  int dimx, int dimy,
  int (*plots_prev)[2], int *attribs_prev, int (*pred_prev)[2],
  int (*plots)[2], int *attribs, int numplots_prev, int numplots,
  int globaltrans[2], int distx, int disty, int win, int var,
  float seuil_ng, float seuil_conf, int fast);

extern float correl(unsigned char *im1, unsigned char *im2, int genre,
  int dimx, int dimy, int pos1[2], int pos2[2], int halfwin, int var,
  float seuil_ng, float res[2]);

extern int readplots (char *nomPlots, int (**plots)[2], char *champsxy,
  char *champspred, int (**pred)[2], char *champattrib, int **attribs);

extern void writematches (char *nomMatches, float (*m1)[2], float (*m2)[2],
  int nummatches, int numcorrect, float *scores, float *confidence,
  char *ligne_de_commande, int width, int height);

extern int checkmatches (float (*m1)[2], float (*m2)[2], float *scores,
  int nummatches, int varmvt, float check);

extern      void    read_file(char *,int ,int ,char *);
extern      void    write_file(char *,int ,int ,char *);
void  read_plots(char *pcFile,int (*piPlots)[2],int *piOri,int *iNbre);
void  write_plots(char *pcFile,int (*piPlots)[2],int *piOri,int *iNbre);
void write_matches(char *, float (*)[2], float (*)[2], int, int, float *,
float *, int, int);
```

```
#endif
```

ANNEXE 1-8

```
/*************************************************************************
*
*
*     MODULE:     type_a.h                        Version     1.0
*
*     PROJET: Estimation de mouvement - copyright Aerospatiale 98
*
*     DESCRIPTION: fichier de definition
*
*     CONTENU:     ,  ,  ,
*
*
*     HISTORIQUE:       creation
*                             20/10/98    by    LEVEQUE     JP          E/SCS/V
*
*************************************************************************
*/
#ifndef     _TYPE_A_H

#define     U1BTIMAG 1
#define U2BTIMAG 2

#define     U1BT unsigned char
#define     U2BT unsigned short

#endif
```

Annexe 2 : Macros et fonctions spécifiques à l'application avec SynDEx v5

Nous trouverons ci-après, le détail de la bibliothèque de macros spécifiques à l'application qui permet d'associer les opérations SynDEx aux fonctions C. Puis, plus loin, nous détaillerons les fonctions C précédemment associées aux opérations.

Bibliothèque de macros spécifiques à l'application

Fichier Algomono.m4x

```
dnl Kamel MEZIANE
dnl Algomono.m4x      10/05/2001
dnl Syndex v5 executive macros specific to application "Algomono"
divert(-1)

divert
/* Kamel MEZIANE, Application "Algomono" SynDEx V5. 10/05/2001*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#define ushort unsigned short
#define uchar  unsigned char
divert(-1)

define(`NBITERATIONS', 1)


###############
# Constantes
#

# constant HEIGHT HEIGHT {int !height}
def(`HEIGHT', `proto_(int!$1)dnl verif du type de HEIGHT
_($1[0]=256;)')

# constant WIDTH WIDTH {int!width}
def(`WIDTH', `proto_(int!$1)dnl verif du type de WIDTH
_($1[0]=256;)')

# constant SEUIL SEUIL {float!seuil}
def(`SEUIL', `proto_(float!$1)dnl verif du type de SEUIL
_($1[0]=20.0;)')

# constant CONF CONF {float!conf}
def(`CONF', `proto_(float!$1)dnl verif du type de CONF
_($1[0]=15.0;)')

# constant DISTX DISTX {int!distx}
def(`DISTX', `proto_(int!$1)dnl verif du type de DISTX
_($1[0]=30;)')

# constant DISTY DISTY {int!disty}
def(`DISTY', `proto_(int!$1)dnl verif du type de DISTY
_($1[0]=30;)')

# constant VAR VAR {int!var}
def(`VAR', `proto_(int!$1)dnl verif du type de VAR
_($1[0]=1;)')
```

```
# constant WIN WIN {int!win}
def(`WIN', `proto_(int!$1)dnl verif du type de WIN
_($1[0]=9;)')

# constant CHECK CHECK {float!check}
def(`CHECK', `proto_(float!$1)dnl verif du type de CHECK
_($1[0]=10.0;)')

# constant VARMVT VARMVT {int !varmvt}
def(`VARMVT', `proto_(int!$1)dnl verif du type de VARMVT
_($1[0]=10;)')




####################
# Fonction Masquage
#

divert
void masquage(uchar *imag, int height, int width, uchar *im);
divert(-1)

# function Mask1  Masquage1 {uchar[65536] ?Imag  int ?height  int ?width  uchar[65536]
!Im}
def(`Masquage1',`ifelse(MGC,LOOP, `proto_(uchar[65536]?$1, int?$2, int?$3,
uchar[65536]!$4)
Ccall_(void, `masquage', uchar *$1, int $2, int $3, uchar *$4)')')

# function Mask2  Masquage2 {uchar[65536] ?Imag  int ?height  int ?width  uchar[65536]
!Im}
def(`Masquage2',`ifelse(MGC,LOOP, `proto_(uchar[65536]?$1, int?$2, int?$3,
uchar[65536]!$4)
Ccall_(void, `masquage', uchar *$1, int $2, int $3, uchar *$4)')')




########################
# Fontion Lecture Image
#

divert
/*void read_file(char *pcFile, int iTailleX, int iTailleY, char *pcBuff);*/
void lire_image(char *Nomimag, int Largeur, int Hauteur, uchar *imagBuff);
divert(-1)



# extio LireIm1 LireImage1 {uchar[65536] !Imag}
def(`LireImage1',`ifelse(MGC,LOOP, `proto_(uchar[65536]!$1)
Ccall_(void, `lire_image', const "image3", const 256, const 256, uchar *$1)')')

# extio LireIm2 LireImage2 {uchar[65536] !Imag}
def(`LireImage2',`ifelse(MGC,LOOP, `proto_(uchar[65536]!$1)
Ccall_(void, `lire_image', const "image4", const 256, const 256, uchar *$1)')')




####################
# Fonction Reduction
#
```

```
divert
void reduction(int height, uchar *im, int width, ushort *imreduc);
divert(-1)


# function Reduc1 Reduction1 {uchar[65536] ?Im  int ?height  int ?width  ushort[16384]
!Imreduc}
def(`Reduction1',`ifelse(MGC,LOOP, `proto_(uchar[65536]?$1, int?$2, int?$3,
ushort[16384]!$4)
Ccall_(void, `reduction', int $2, uchar *$1, int $3, ushort *$4)')')

# function Reduc2 Reduction2 {uchar[65536] ?Im  int ?height  int ?width  uchar[16384]
!Imreduc}
def(`Reduction2',`ifelse(MGC,LOOP, `proto_(uchar[65536]?$1, int?$2, int?$3,
ushort[16384]!$4)
Ccall_(void, `reduction', int $2, uchar *$1, int $3, ushort *$4)')')




##########################
# Fonction Pre-estimation
#

divert
void preestim(int height, ushort *im1reduc, ushort *im2reduc, float seuil, int width,
int *globaltrans);
divert(-1)


# function Preestim  Preestimation { int ?height ushort[16384] ?im1reduc ushort[16384]
?im2reduc  float ?seuil
#                                int ?width  int[2] !globaltrans}
def(`Preestimation',`ifelse(MGC,LOOP, `proto_(int?$1, ushort[16384]?$2,
ushort[16384]?$3, float?$4, int?$5, int[2]!$6)
Ccall_(void, `preestim', int $1, ushort *$2, ushort *$3, float $4, int $5, int
*$6)')')




########################
# Fontion Lecture Plots
#


divert
void read_plots_SansOri(char *pcFile, int (*piPlots)[2], int *iNbre);
/*void lire_plots_SO(char *NomPlots, int *Liste, int *Nbpts);*/
divert(-1)


# extio Plots1 LirePlots1 {int !numplots int[1000] !plots}
def(`LirePlots1',`ifelse(MGC,LOOP, `proto_(int!$1, int[1000]!$2)
Ccall_(void, `read_plots_SansOri', const "PlesseySimuNew3.txt", int *$2, int *$1)')')

# extio Plots2 LirePlots2 {int !numplots int[1000] !plots}
def(`LirePlots2',`ifelse(MGC,LOOP, `proto_(int!$1, int[1000]!$2)
Ccall_(void, `read_plots_SansOri', const "PlesseySimuNew4.txt", int *$2, int *$1)')')



########################
# Fontion Appariement
#
```

```
divert
void appariement(float conf, int distx, int disty, int *globaltrans,  int height,
              uchar *im1, uchar *im2, int numplots1, int numplots2, int *plots1,
              int *plots2, float seuil, int var, int width, int win,
              float *confidence, float *m1,float *m2, int *numatches, float *scores);
divert(-1)

# function Appar Appariement { float ?conf int ?distx int ?disty int[2] ?globaltrans
int ?height uchar[65536] ?im1 uchar[65536] ?im2 int ?numplots1 int ?numplots2
int[1000] ?plots1 int[1000] ?plots2 float ?seuil int ?var int ?width int ?win
float[500] !confidence float[1000] !m1 float[1000] !m2 int !numatches float[500]
!scores}
def(`Appariement',`ifelse(MGC,LOOP, `proto_(float?$1, int?$2, int?$3, int[2]?$4,
int?$5, uchar[65536]?$6, uchar[65536]?$7, int?$8, int?$9,int[1000]?$10, int[1000]?$11,
float?$12, int?$13, int?$14, int?$15, float[500]!$16, float[1000]!$17,
float[1000]!$18, int!$19, float[500]!$20)
Ccall_(void,`appariement', float $1, int $2, int $3, int *$4, int $5, uchar *$6, uchar
*$7, int $8, int $9, int *$10, int*$11, float $12, int $13, int $14, int $15, float
*$16, float *$17, float *$18, int *$19, float *$20)')')



################################################
# Fonction Filtrage des Appariements incorrects
#

divert
void filtr_appar(float *m1, float *m2, float *scoresi, int nummatches, int varmvt,
float check, float *scoreso, int *numcorrect);
divert(-1)

# function FiltrApparInc  FiltrageAppar { float[1000] ?m1 float[1000] ?m2  float[500]
?scores int ?nummatches int ?varmvt float ?check float !scores int !numcorrect}
def(`FiltrageAppar',`ifelse(MGC,LOOP, `proto_(float?$1, float[1000]?$2,
float[1000]?$3, int?$4, float[500]?$5, int?$6, int!$7, float[500]!$8)
Ccall_(void, `filtr_appar', float *$2, float *$3, float  *$5, int $4, int $6, float
$1, float *$8, int *$7)')')



#####################################
# Fonction Ecriture des appariements
#

divert
void write_matches(char *pcFile, float *m1, float *m2, int nummatches, int numcorrect,
float *scores, float *confidence, int width, int height);
divert(-1)


# extio EcrireAp EcrireAppar {float[500] ?confidence int ?height float[1000] ?m1
float[1000] ?m2 int ?numcorrect
#                         int ?nummatches float[500] ?scores int ?width}
def(`EcrireAppar',`ifelse(MGC,LOOP, `proto_(float[500]?$1, int?$2, float[1000]?$3,
float[1000]?$4, int?$5, int?$6, float[500]?$7, int?$8)
Ccall_(void, `write_matches', const "Appariements.asc", float *$3, float *$4, int $6,
int $5, float *$7, float *$1, int $8, int $2)')')



####################
# Fonction Ecriture
#
```

```
divert
/*void write_file(char *pcFile, int iTailleX, int iTailleY, char *pcBuff);*/
divert(-1)


# extio EcrireIm1 EcrireImage1 {ushort[16384] ?ImagRed}
def(`EcrireImage1',`ifelse(MGC,LOOP, `proto_(ushort[16384]?$1)
Ccall_(void, `write_file', const "image3red", const 128, const 128, ushort *$1)')')

# extio EcrireIm2 EcrireImage2 {ushort[16384] ?ImagRed}
def(`EcrireImage2',`ifelse(MGC,LOOP, `proto_(ushort[16384]?$1)
Ccall_(void, `write_file', const "image4red", const 128, const 128, ushort *$1)')')


divert`'dnl ---------------- fin du fichier  ------------------
```

Fonctions spécifiques à l'application

Certaines des fonctions suivantes font appel à des fonctions décrites dans l'annexe 1.

```
/*************************************************************************
*
* lire_image() - Lecture de l'image
*
*************************************************************************/

void lire_image(char *Nomimag, int Largeur, int Hauteur, uchar *imagBuff)
{
  read_file(Nomimag, Largeur, Hauteur, (char *)imagBuff);
}


/*************************************************************************
*
* masquage() - Masquage de l'image
*
*************************************************************************/

void masquage(uchar *imag, int height, int width, uchar *imagmask)
{
  int i, j;

  for ( j=0 ; j<height ; j++ )
     {
       for ( i=0; i<width ; i++ )
         {
           imagmask[j*width+i] = imag[j*width+i] & 0x0ff;
         }
     }
}


/*************************************************************************
*
* reduction() - Reduction de l'image
*
*************************************************************************/

void reduction(int height, uchar *im, int width, ushort *imreduc)
{
  /*  height  :   entree hauteur de l'image  */
  /*  *im     :   entree image               */
  /*  width   :   entree largeur de l'image  */
  /*  *imreduc:   sortie image reduite       */

  int i, j, k;

  for (k = 0; k< (height>>2)*(width>>2); k++)
     {
       imreduc[k] = 0;
     }

  for(j = 0; j<height; j++)
     {
       for (i = 0; i<width; i++)
          {
            k=(j >> 1)*(width >> 1)+(i >> 1);
            imreduc[k] += im[j*width+i];
          }
     }
}
```

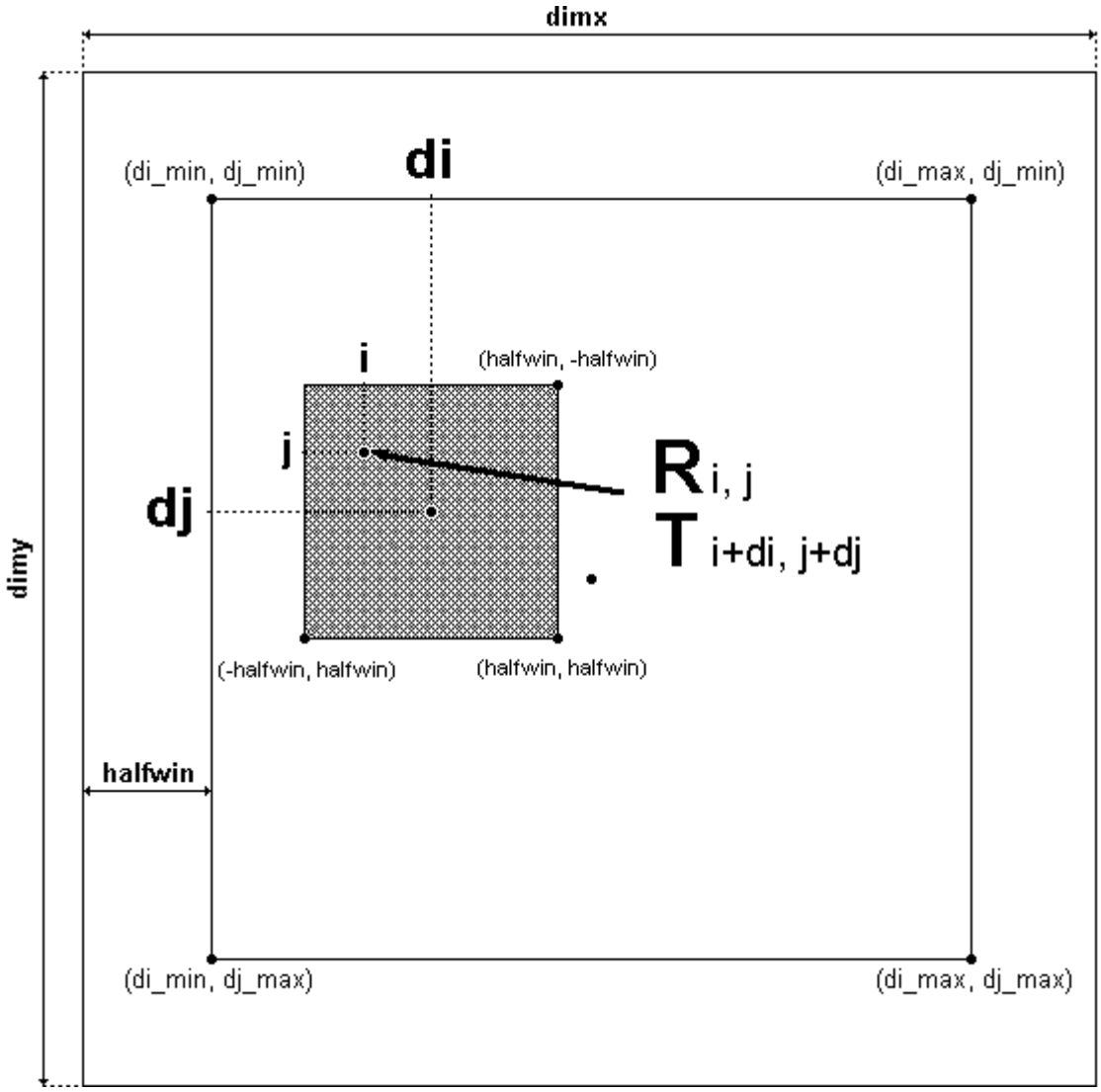
```
/*********************************************************************
*
* preestim() - Pre-estimation de la translation
*
*********************************************************************/

void preestim(int height, ushort *im1reduc, ushort *im2reduc, float seuil, int width,
int *globaltrans)
{

  int   center[2];
  float res[2];
  float seuil_ng;

  center[0]= width>>2;
  center[1]= height>>2;

  globaltrans[0]= 0;
  globaltrans[1]= 0;

  seuil_ng= seuil*4;

#ifdef PRINT
       printf("\n************************************************\n");
       printf("\nProcessing pre-estimation...\n\n");
#endif

  if ( correl((char *)im1reduc,(char *)im2reduc, U2BTIMAG, width>>1, height>>1,
center, center, width>>4, width>>2, seuil_ng, res))
    {
      globaltrans[0]= (int) (res[0] - center[0]) * 2;
      globaltrans[1]= (int) (res[1] - center[1]) * 2;
    }


#ifdef PRINT
       printf(" center[0] x: %d\n",center[0]);
       printf(" center[1] y: %d\n\n",center[1]);
       printf(" res[0] x: %f\n",res[0]);
       printf(" res[1] y: %f\n\n",res[1]);
       printf(" Translation globale en x: %d\n",globaltrans[0]);
       printf(" Translation globale en y: %d\n",globaltrans[1]);
       printf("\n************************************************\n");
#endif
}
```

```
/*********************************************************************
*
* appariement() - Appariement des points caracteristiques
*
*********************************************************************/

void appariement(float conf, int distx, int disty, int *globaltrans, int height,
              uchar *im1, uchar *im2, int numplots1, int numplots2, int *plots1,
              int *plots2, float seuil, int var, int width, int win,
              float *confidence, float *m1,float *m2, int *numatches, float *scores)
{
  int genre=1;
  int (*pred)[2] = NULL;
  int fast=1;

  int i;
#ifdef PRINT
       printf("\nNombre de plots des deux listes:\n\n");
       printf(" Liste1 (numplots1): %d\n", numplots1);
       printf(" Liste2 (numplots2): %d\n", numplots2);
       printf("\n*************************************************\n");
#endif

#ifdef  KTEST
        printf("\nAffichage des premiers pts des listes\n\n");
      for (i=0 ; i<10; i=i+2)
         {
           printf("plots1 x= %d", plots1[i]);
           printf(", y= %d\n", plots1[i+1]);
           printf("plots2 x= %d", plots2[i]);
           printf(", y= %d\n\n", plots2[i+1]);
         }
      printf("\n*************************************************\n");
#endif


#ifdef ORIENTATION
#ifdef    PRINT
          printf("\nAppariement des pts caracteristiques...\n");
           printf("\nProcessing with orientation !!!\n\n");
#endif

     *numatches = matcheplots(im1, im2, genre, width, height, plots1, origrad1, pred,
plots2, origrad2, numplots1, numplots2, globaltrans, distx, disty, win, var, seuil,
conf, fast, confidence, m1, m2, scores);
#else
#ifdef    PRINT
          printf("\nAppariement des pts caracteristiques...\n");
           printf("Processing without orientation !!!\n\n");
#endif

     *numatches = matcheplots(im1, im2, genre, width, height, plots1,     NULL, pred,
plots2,     NULL, numplots1, numplots2, globaltrans, distx, disty, win, var, seuil,
conf, fast, confidence, m1, m2, scores);
#endif

#ifdef  PRINT
      printf(" Nb de pts apparies (nummatches): %d\n", *numatches);
      printf("\n*************************************************\n");
#endif

#ifdef  KTEST
      printf("\nAffichage des premiers elements de Listes\n\n");
        for (i=0 ; i<10; i++)
          {
            int j=2*i;

            printf(" ---------------------------------\n");
```

```
            printf(" conf[%d]= %f\n", i, confidence[i]);

            printf(" m1[%d][0] = %f", i, m1[j]);
            printf(" m1[%d][1] = %f\n", i, m1[j+1]);
            printf(" m2[%d][0] = %f", i, m2[j]);
            printf(" m2[%d][1] = %f\n", i, m2[j+1]);

            printf(" scores[%d] = %f\n", i, scores[i]);
          }
       printf("\n*************************************************\n");
#endif
}



/*********************************************************************
*
* filtr_appar() - Filtrage des appariements incorrects
*
*********************************************************************/

void filtr_appar(float *m1, float *m2, float *scoresi, int nummatches, int varmvt,
float check, float *scoreso, int *numcorrect)
{
  int i;

  for ( i=0; i<nummatches ; i++)
    {
      scoreso[i] = scoresi[i];
    }

#ifdef PRINT
  printf("\nFiltrage des appariements incorrects...\n");
#endif

  if (check > 0)
    {
      *numcorrect = checkmatches(m1, m2, scoreso, nummatches, varmvt, check);

#ifdef  KTEST
       printf("\n numcorrect intermediaire: %d\n\n", *numcorrect);

       for (i=0 ; i<20; i++)
          {
            printf(" ---------------------------------------\n");
            printf(" scoresi[%d] = %f\n", i, scoresi[i]);
            printf(" scoreso[%d] = %f\n", i, scoreso[i]);
          }
#endif
    }

  *numcorrect= fComputeMedian (m1, m2, scoreso, nummatches);

#ifdef  PRINT
       printf("\n Nb d'appariements correct(numcorrect): %d\n", *numcorrect);
       printf("\n*************************************************\n");
#endif
#ifdef  KTEST
       printf("\nAffichage de la liste des scores\n\n");
         for (i=0 ; i<nummatches; i++)
           {
             printf(" scoreso[%d] = %f\n", i, scoreso[i]);
           }
       printf("\n*************************************************\n");
#endif

}
```

Annexe 3 : Principe de la corrélation

La corrélation est une méthode employée pour rechercher une forme caractéristique dans une image en faisant l'hypothèse d'un déplacement en translation. La formulation de la corrélation que nous utilisons est celle des valeurs absolues des différences.



Principe de la corrélation

dimx et dimy	: dimensions de notre image.
halfwin	: demi-largeur du motif de corrélation (en grisé sur la figure)
di_min, di_max	: déplacements autorisés en abscisse
dj_min, dj_max	: déplacements autorisés en ordonnée
$R_{i, j}$	: pixel (i, j) du motif de corrélation
$T_{i+di, j+dj}$	: pixel (i, j) de la zone de recherche

Nous balayons la zone délimitée par di et dj avec le centre du motif de corrélation. Pour chaque couple (di, dj) nous effectuons la corrélation (des valeurs absolues des différences) et nous obtenons un score. A l'issue du balayage complet de la zone de recherche, nous cherchons le point de l'image réduite 2 correspondant au score minimum (pic de corrélation).

Corrélation des valeurs absolues des différences :

$$C_{di,dj} = \sum_{i=-halfwin, j=-halfwin}^{i=halfwin, j=halfwin} \left| R_{i,j} - T_{i+di, j+dj} \right|$$

On recherche le pic de corrélation :

$$C = \min_{di,dj} \left(C_{di,dj} \right)$$

Annexe 4 : Fonctions spécifiques à l'application avec SynDEx v6

Ce tableau met en correspondance les définitions de notre algorithme, les références à ces définitions ainsi que les fonctions C associées.

Les fonctions écrites en gras sont disponibles dans la suite de cette annexe. Les autres sont disponibles soit dans l'annexe 1 (nom des fonctions est souligné, ANNEXE 1-6), soit dans l'annexe 2.

Définitions	Références	Fonctions
LireImage1	LireIm1	`lire_image()`
LireImage2	LireIm2	
Masquage	Mask1	`masquage()`
	Mask2	
Reduction	Reduc1	`reduction()`
	Reduc2	
CalculParamPreestim	ParametresPreestim	**`calcparampreestim()`**
CalcDeplac	CalculDeplacements	**`calcdeplac()`**
MotifCorrelIm1	MotifCorrelationIm1	**`motifcorrelim1()`**
GenVectIndices	GenereVecteurIndices	**`genvectindic()`**
CorrelationLig	CorrelationLigneP	**`correlig()`**
CoordsPicCorrel	CoordonneesPicCorrel	**`coordpiccorrel()`**
TransGlobale	TranslationGlobale	**`transglobale()`**
LirePlots1	Plots1	`read_plots_SansOri()`
LirePlots2	Plots2	
ElimPtsBord	ElimBord1	**`elimptsbord()`**
	ElimBord2	
ApparListe2	AppariementEtape2	**`apparetape2()`**
FenPtsCaract	FentragePtsCaracteristiques	**`fenptscaract()`**
ScoreMinEtConf	ScoreMinimumEtConfiance	**`scoreminconf()`**
SelecPtsAppar	SelectionPtsAppar	**`selectptsappar()`**
FiltrageAppar	FiltrApparInc	`filtr_appar()`
EcrireAppar	EcrireAp	`write_matches()`

Les opérations dont le nom est écrit en gras sont représentées, dans la suite de cette annexe, selon leur appartenance aux opérations de pré-estimation ou d'appariement.

Preestim

```
/*************************************************************************
*
* calcparampreestim() - Calcul des paramètres de la pré-estimation
*
*************************************************************************/

void calcparampreestim(int height, int width, float seuil, int *dimx, int *dimy,
int *center, int *halfwin, int *var, float *seuil_ng)
{
  center[0] = width>>2;
  center[1] = height>>2;

  *dimx = width>>1;
  *dimy = height>>1;

  *halfwin = width>>4;
  *var= width>>2;

  *seuil_ng= 4*seuil;
}


/*************************************************************************
*
* calcdeplac() - Calcul des déplacements
*
*************************************************************************/

void calcdeplac(int var, int dimy, int dimx, int *pos2, int halfwin,
             int *cond, lint *di_min, lint *di_max, lint *dj_min, lint *dj_max)
{

  int scoresize;

  scoresize = 10000;
  *cond = 1;
  var += 1;

  if (pos2[0] - halfwin - var >= 0)
    { *di_min = -var; }
  else
    { *di_min = halfwin - pos2[0]; }


  if (pos2[0] + halfwin + var < dimx)
    { *di_max = var; }
  else
    { *di_max = dimx-1 - halfwin - pos2[0]; }


  if (pos2[1] - halfwin - var >= 0)
    { *dj_min = -var; }
  else
    { *dj_min = halfwin - pos2[1]; }


  if (pos2[1] + halfwin + var < dimy)
    { *dj_max = var; }
  else
    { *dj_max = dimy-1 - halfwin - pos2[1]; }


  if ((*di_max-*di_min+1) < 3 || (*dj_max-*dj_min+1) < 3)
    { *cond=0; }
```

```
  /*
   * Alloue un tableau pour stocker les valeurs de correlation a interpoler.
   * Optimisation : on conserve l'espace alloue aux appels precedants.
   */

  if ((*dj_max-*dj_min+1)*(*di_max-*di_min+1) > scoresize)
    { printf("Tableau des scores excedent la taille alloue: %d\n",
             (*dj_max-*dj_min+1)*(*di_max-*di_min+1));
      *cond = 0; }

}


/*****************************************************************
*
* genvectindic() - Génération de vecteur d'indices
*
*****************************************************************/

void genvectindic(lint di_min, lint di_max, lint dj_min, lint dj_max,
               lint *vectdi, int *nbdi, lint *vectdj, int *nbdj)
{
 int i, compte;

 *nbdi = di_max - di_min +1;
 *nbdj = dj_max - dj_min +1;

 compte = 0;
 for ( i=di_min ; i<=di_max ; i++, compte++)
    {
      vectdi[compte] = i;
    }
 compte = 0;
 for ( i=dj_min ; i<=dj_max ; i++, compte++)
    {
      vectdj[compte] = i;
    }
}


/*****************************************************************
*
* motifcorrelim1() - Motif de corrélation
*
*****************************************************************/

void motifcorrelim1(ushort *im1, int *pos1, int size, int halfwin, int dimx,
                    ushort *MCIm1)
{
 int i, j, numpts;
 int win, nextline;
 ushort *p;

 size = 1;
 numpts = 0;
 win = 2*halfwin +1;
 nextline = (dimx - win) * size;

 p = im1 + ( pos1[0]-halfwin  +  dimx*(pos1[1] - halfwin) ) * size;

 for (j = -halfwin ; j <= halfwin; j++, p +=nextline)
    {
      for (i = -halfwin ; i <= halfwin; i++, p +=size, numpts++)
        {
          MCIm1[numpts] = *p;
        }
    }
}
```

```
/*************************************************************************
*
* correlig() - Correlation en ligne
*
*************************************************************************/

void correlig(ushort *MCIm1, int *pos1, int *pos2, ushort *im2, int size,
              int halfwin, int dimx, int *vectdi, int nbdi, int *vectdj,
              int nbdj, int nbrepapp1, int taillevect, lint *scores, lint *deplacs)
{

 int win, nextline, ligne, colonne;
 int numpts;
 int i, j, di, dj;
 ushort *p;
 lint   *pscores;

 size = 1;
 win = 2*halfwin +1;
 nextline = (dimx - win) * size;

 ligne = 0;
 for (dj= vectdj[0] ; dj<=vectdj[taillevect-1] ; dj++, ligne++)
       {
         colonne=0;
      for (di= vectdi[0] ; di<=vectdi[taillevect/nbrepapp1-1] ; di++, colonne++)
             {

          p = im2 + ( pos2[0]+di-halfwin  +  dimx*(pos2[1]+dj-halfwin) ) * size;
               pscores = scores + colonne + taillevect*ligne ;
               deplacs[2*(colonne + taillevect*ligne)] = di;
               deplacs[2*(colonne + taillevect*ligne)+1] = dj;

          numpts = 0;
          for (j = -halfwin ; j <= halfwin; j++, p +=nextline)
                    {
                      for (i = -halfwin ; i <= halfwin; i++, p +=size, numpts++)
                            {
                     *pscores += abs( MCIm1[numpts] - *p);
                            }
                    }
             }
       }
}



/*************************************************************************
*
* coordpiccorrel() - Coordonnées du pic de corrélation
*
*************************************************************************/

void coordpiccorrel(lint *scores, lint *deplacs, int *pos2, int halfwin,
                    int nbdi, int nbdj, lint di_min, lint di_max, lint dj_min,
                    lint dj_max, float seuil_ng, float *res, float *score)
{

   lint    imin=0, jmin=0;
   lint    i, j;
   lint    *pscore, minscore;

   lint    z1, z2, z3, z4, z5, a2, b2, c2, d2;
   float   dx = 0, dy = 0;
   float   ret;

   int win;
```

```
/*
 * Recherche du pic de correlation discret
 */

 minscore = -1;

 for (j = dj_min, pscore=scores ; j <= dj_max ; j++)
    {
      for (i = di_min; i <= di_max; i++, pscore++)
         {
           if (minscore < 0 || *pscore < minscore)
             {
               minscore = *pscore;
                imin = i;
                jmin = j;
             }
         }
    }

 win = 2*halfwin +1;

/*
 * Si le pic de correlation est sur un bord, ce n'est probablement pas
 * un minimum local, donc on invalide l'appariement.
 * Si la valeur de correlation est trop elevee, invalide l'appariement.
 */

 if (minscore < 0 ||
     imin == di_min || imin == di_max ||
     jmin == dj_min || jmin == dj_max ||
     minscore > seuil_ng * win*win)
   {
     minscore = -1;
     res[0] = 0;
     res[1] = 0;
   }
 else
   {
     /*
      * Resultat de la correlation discrete.
      */

     res[0] = (float)(pos2[0] + imin);
     res[1] = (float)(pos2[1] + jmin);


     /*
      * Determination de la position subpixellique du pic de correlation.
      * On approxime les valeurs de correlation par une fonction quadratique
      * sur un voisinage en 4-connexite :
      *
      *    z1
      * z2 z3 z4
      *    z5
      *
      *  z = a x^2 + b y^2 + c x + d y + f
    *
      * d'ou :
      *
      *  a = 1/2 (z2 + z4 - 2 z3)
      *  b = 1/2 (z1 + z5 - 2 z3)
      *  c = 1/2 (z4 - z2)
      *  d = 1/2 (z5 - z1)
      *  f = z3
      *
      * Position du minimum :
      *
    *  x = - c / 2a
```

```
         *  y = - d / 2b
         */

        pscore = scores + imin-di_min + (jmin-dj_min-1)*(di_max-di_min+1);
        z1 = *pscore;
        pscore += di_max-di_min+1-1;
        z2 = *pscore;
        pscore ++;
        z3 = *pscore;
        pscore ++;
        z4 = *pscore;
        pscore += di_max-di_min+1-1;
        z5 = *pscore;

        a2 = z2 + z4 - 2 * z3;
        b2 = z1 + z5 - 2 * z3;
        c2 = z4 - z2;
        d2 = z5 - z1;

        if (a2 == 0)
          { dx = 0; }
        else
          { dx = - c2 / (float)(2 * a2); }

        if (b2 == 0)
          { dy = 0; }
        else
          { dy = - d2 / (float)(2 * b2); }
      }

        res[0] += dx;
        res[1] += dy;

  *score = (float)minscore;

}


/*************************************************************************
*
* transglobale() - Calcul de la translation globale
*
*************************************************************************/

void transglobale(float *res, float score, int *center, int *globaltrans)
{

 globaltrans[0] = 0;
 globaltrans[1] = 0;

 if (score != -1)
   {
     globaltrans [0] = (int) (res[0]-center[0])*2;
     globaltrans [1] = (int) (res[1]-center[1])*2;
   }
}
```

Appar

```
/*********************************************************************
*
* elimptsbord() - Elimination des points se trouvant sur les bords
*
*********************************************************************/

void elimptsbord(int *inplots, int innumplots, int height, int width,
                 int halfwin, int *outnumplots, int *outplots)
{
  int i, nombre;

  nombre = 0;
  for (i=0; i< 2*innumplots; i=i+2)
     {
       if( inplots[i] < halfwin || inplots[i] >= width - halfwin ) continue;
       if( inplots[i+1] < halfwin || inplots[i+1] >= height - halfwin ) continue;

       outplots[2*nombre] = inplots[i] ;
       outplots[2*nombre+1] = inplots[i+1] ;
       nombre++;
     }

  *outnumplots = nombre;
}



/*********************************************************************
*
* fenptscaract() - Fenêtrage des points caractéristiques
*
*********************************************************************/

void fenptscaract(int numplots_prev, int *plots_prev, int distx, int *globaltrans,
                  int disty, int numplots, int *plots, int max_fenet,
                    int *sousnumplots, int *sousplots)
{
  int i, j;
  lint numero1, numero2;
  int dx, dy;

  numero1 = 0;
  for (i=0 ; i<2*numplots_prev; i=i+2, numero1++)
     {
       numero2 = 0;
       for (j=0 ; j<2*numplots || (numplots<0 && j==0); j=j+2)
          {
            dx = plots[j]   - plots_prev[i]   - globaltrans[0];
            dy = plots[j+1] - plots_prev[i+1] - globaltrans[1];

            if (abs(dx) > distx || abs(dy) > (disty >= 0 ? disty : distx)  ||
numero2 >= max_fenet) continue;

            sousplots[j+numero1*max_fenet] = plots[j];
            sousplots[j+1+numero1*max_fenet] = plots[j+1];
            numero2++;
          }
       sousnumplots[numero1] = numero2;
     }
}
```

```
/*************************************************************************
*
* apparetape2() - Appariement Etape2 (pour chaque pt de la sous liste 2)
*
*************************************************************************/

void apparetape2(int fast, uchar *im_prev, int numplots_prev, int (*plots_prev)[2],
                    int *sousnumplots, int (*sousplots)[2], uchar *im, int genre,
                    int halfwin, int win, int dimx, float seuil_ng, int dimy,
                    int var, int max_plot, int max_fenet, int nbrepapp1,
                    float *scores, float (*res)[2])
{
  int i, j, maxi_plot;
  float result[2];
  float score;
  int condit[75*40];
  maxi_plot = max_plot/nbrepapp1;

  for (j=0 ; j<maxi_plot ; j++)
     {
       for (i=0 ; i<max_fenet ; i++)
          {
            condit[i+j*max_fenet] = 1;
          }
     }

  if (fast)
    {
      uchar *p1, *p2;
      int nextline;
      int ii,jj;

      for (j=0 ; j<maxi_plot ; j++)
        {
           for (i=0 ; i<max_fenet ; i++)
              {
                nextline = (dimx - win) * size;

p1 = im_prev + (plots_prev[j][0] - halfwin + dimx*(plots_prev[j][1] - halfwin)) *
size;
p2 = im + (sousplots[i+j*max_fenet][0]-halfwin + dimx *(sousplots[i+j*max_fenet][1]
- halfwin)) * size;

                score = 0;

         for (jj = -halfwin ; jj <= halfwin; jj++, p1 += nextline,  p2 += nextline)
             {
                for (ii = -halfwin ; ii <= halfwin; ii++, p1 += size, p2 += size)
                    {
                      score += abs( *p2 - *p1 );
                    }
             }

                 if (score > 1.5 * seuil_ng * win*win)
                   {
                     condit[i+j*max_fenet]= 0;
                   }
               }
        }
    }

 for (j=0 ; j<maxi_plot ; j++)
    {
      for (i=0 ; i<max_fenet ; i++)
         {
           if (condit[i+j*max_fenet] ==0)
             {
                      lesscores[i+j*max_fenet] = -1;
```

```
                    }
                  else
                    {
                score = correl(im_prev, im, size, dimx, dimy, plots_prev[j],
sousplots[i+j*max_fenet], halfwin, var, seuil_ng, result);

                lesscores[i+j*max_fenet] = score;
                lesres[i+j*max_fenet][0] = result[0];
                lesres[i+j*max_fenet][1] = result[1];
               }
           }
      }
}


/*********************************************************************
*
* scoreminconf() - Détermination du score minimum et de la confiance
*
*********************************************************************/
void scoreminconf(int (*plots_prev)[2], int *sousnumplots, float *scores,
                  float (*res)[2], int win, float seuil_conf, int maxplot,
                    int nbrepapp1, float (*ptsdepart)[2], float (*ptsarrivee)[2],
                    float *minscores, float *confs, int *matches)
{

  int i, j;
  int maxi_plot;
  float minscore, secondminscore, conf;

  maxi_plot = max_plot/nbrepapp1;
  for (i=0 ; i<maxi_plot ; i++)
     {
       minscores[i]= -1;
     }
  for (j=0 ; j<maxi_plot ; j++)
     {
       if (sousnumplots[j] <= 0) continue;
       secondminscore = -1;
       minscore = -1;
       for (i=0 ; i<40 ; i++)
          {
            if (scores[i+j*40] <0) continue;
            if (minscore<0 || scores[i+j*40]<minscore)
              {
                secondminscore = minscore;
                minscore = scores[i+j*40];
                matches[j] = i;
                ptsdepart[j][0] = (float) plots_prev[j][0];
                ptsdepart[j][1] = (float) plots_prev[j][1];
                ptsarrivee[j][0] = res[i+j*40][0];
                ptsarrivee[j][1] = res[i+j*40][1];
             }
            else if (scores[i+j*40]<secondminscore)
                   {
                     secondminscore = scores[i+j*40];
                   }
          }
       conf = (secondminscore < 0)? 1:(secondminscore - minscore)/secondminscore;
       conf *= 100;

       if (conf < seuil_conf) continue;
       minscore /= win*win;
       minscores[j] = minscore;
       confs[j] = conf;
     }
}
```

```
/*************************************************************************
*
* selectptsappar() - Sélection des points appariés
*
*************************************************************************/

void selectptsappar(float (*ptsdepart)[2], float (*ptsarrivee)[2],
                    float *minscores, float *confs, int *matches,
                    int max_plot,float *scores, float (*m1)[2], float (*m2)[2],
                    int *nummatches, float *confidence, )
{
  int i, j, nombre;
  float scoreni;

  nombre = 0;

  for (i=0 ; i<max_plot ; i++)
    {
        if (minscores[i] == -1) continue;
      scoreni = minscores[i];

      for (j=i+1; j<max_plot || i==max_plot; j++)
             {
          if (minscores[j] == -1) continue;
               if (matches[i] == matches[j])
                    {

                      if (scoreni > minscores[j])
                           {
                  minscores[i] = -1;
                             scoreni = minscores[j];
                           }
                      else
                           {
                  minscores[j] = -1;
                           }
                    }
             }
       }

  for (i=0 ; i<max_plot ; i++)
     {
        if (minscores[i] == -1) continue;

      m1[nombre][0] = ptsdepart[i][0];
      m1[nombre][1] = ptsdepart[i][1];

      m2[nombre][0] = ptsarrivee[i][0];
      m2[nombre][1] = ptsarrivee[i][1];

      scores[nombre] = minscores[i];
      confidence[nombre] = confs[i];

      nombre++;
     }
  *nummatches = nombre;
}
```

Annexe 5 : Chronométrage des fonctions associées aux opérations

Les conditions de chronométrage sont les suivantes :

- La dimension des images d'entrées est de 256 pixels * 256 pixels,
- le nombre de points constituant chaque liste de points caractéristiques extrait de chacune des images est fixé à 300,
- le nombre de points des sous-listes de points caractéristiques extraits de l'image 2 est fixé à 40,
- le nombre d'appariements, présent en sortie de la fonction d'appariement, est fixé à 200.

Nous avons chronométré nos fonctions (Annexe 4) avec les données précédentes, afin d'obtenir la durée d'exécution maximale de l'opération associée à la fonction. Nous obtenons, pour chacune des opérations, les résultats inscrits dans le tableau suivant. Les durées d'exécution sont exprimées en µs (unité de temps).

Opérations		Spécification hiérarchique			
		Niveau 1		Niveau 2	
LireImage1	5921,675				
LireImage2	5921,675				
Masquage	5337,3				
Reduction	14144,6				
Preestimation	2199160,725	CalculParamPreestim	1,5		
		CorrelFinePreestim	2199158,075	CalcDeplac	3046,55
				MotifCorrelIm1	147,4
				GenVectIndices	6
				CorrelationLig [4]	548463,325
				CoordsPicCorrel	2104,825
		TransGlobale	1,15		
LirePlots1	35220,125				
LirePlots2	35220,125				
Appariement	27780832,48	ElimPtsBord	256,55		
		ApparListe1 [4]	6942894,85	ApparListe2 [4]	1730051,775
				FenPtsCaract	20544,975
				ScoreMinEtConf	2142,775
		SelecPtsAppar	8739,775		
		CalculHalfwin	0,2		
FiltrageAppar	407,4				
EcrireAppar	2091				

Total de l'algorithme
30103739

Pour maintenir la proportionnalité existante entre toutes les durées d'exécution des opérations et tous les temps de communication, nous avons multiplié par 35 les durées d'exécution de nos opérations et introduit ces nouvelles valeurs dans le tableau suivant. Les nouvelles durées d'exécution seront réinjectées dans SynDEx pour simulation.

Opérations	
LireImage1	207259
LireImage2	207259
Masquage	186805
Reduction	495061
Preestimation	76970626
LirePlots1	1232704
LirePlots2	1232704
Appariement	972329137
FiltrageAppar	14259
EcrireAppar	73185

Spécification hiérarchique		
Niveau 1		Niveau 2
CalculParamPreestim	52	
CorrelFinePreestim	76970534	CalcDeplac 106629 MotifCorrelIm1 5159 GenVectIndices 210 CorrelationLig [4] 19196217 CoordsPicCorrel 73668
TransGlobale	40	
ElimPtsBord	8979	
ApparListe1 [4]	243001320	ApparListe2 [4] 60551812 FenPtsCaract 719074 ScoreMinEtConf 74998
SelecPtsAppar	305892	
CalculHalfwin	7	

Total de l'algorithme
1053630865

L'unité de temps des durées d'exécution des opérations est à présent de 28,57 ns.