Une méthode de parallelisation des Algorithmes de traitement d'images

Abderrazak HENNI *, Rachida SAOULI **

* Institut National de formation en Informatique (I.N.I) BP 68 M
16270 Oued-Smar ALGER ALGERIE
e-mail: henni@ist.cerist.dz
** Université de Biskra BP 145 RP 07000 BISKRA ALGERIE

I. LE TRAITEMENT D'IMAGES ET LE PARALLELISME

Etat de l'art.

L'objectif de notre travail est de définir une méthode de parallélisation des algorithmes séquentiels du traitement d'images. Dans ce cadre nous spécifions d'abord la nature des traitements appliqués ensuite nous présentons brièvement le parallélisme appliqué et nous terminons par une discussion générale des performances requises.

1. Nature des algorithmes:

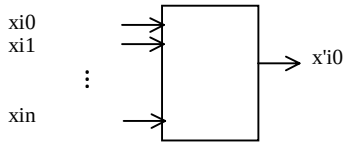
Le traitement d'images concerne diverses manipulations sur des images existantes afin d'extraire des informations pertinentes. Chaque système de traitement d'images applique sur l'image une séquence de traitement répartie principalement en deux niveaux:

a) Le bas niveau: les algorithmes de bas niveau, tel que la détection de contours, traitent directement des matrices images afin d'extraire certaines caractéristiques. Les traitements comportent une grande régularité des opérations et la structure des données manipulées ne se trouve pas modifiée.

b) Le haut niveau: les algorithmes de haut niveau, tel que le traitement des contours, analysent l'image et donnent une description de la scène renfermant en général la recon- naissance et la classification des objets. Des traitements généralement itératifs manipulent des structures de données variées (graphe, réseau, liste chaînée, ..) qui sont gérés dynamiquement (création, suppression de nœuds, ...)

2. Application du parallélisme:

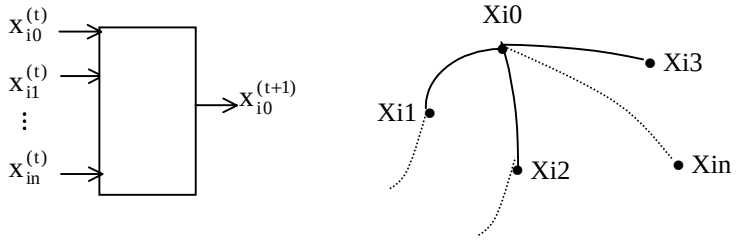
2.1. Les opérateurs de bas niveau:



L'opérateur élémentaire fait correspondre à xi0 le pixel x'i0 par la fonction F(xi0, xi1, ... xin)

Les opérateurs point à point correspondent à n =1 tandis que le opérateur sur le voisinage correspondent à n > 7.
La parallélisation consiste à appliquer un opérateur simultanément sur tous les pixels. Un modèle synchrone convient parfaitement, mettant en œuvre un parallélisme temporel (architecture pipeline) ou un parallélisme spatial qui est en fait un parallélisme de données(architecture SIMD caractérisée par une granularité fine)

2.2 les opérateurs de haut niveau:



En chaque nœud Xi0 du graphe ou réseau représentatif d'une image, on connaît le voisinage Vi0 =(Xi0, ... ,Xi1).
L'application en parallèle de l'opérateur de haut niveau consiste à assigner un processeur à chacun des nœuds puis à appliquer une règle de type $x_{i0}^{(t+1)} = F(x_{i0}^{(t)} , ... x_{in}^{(t)})$. Dans ce sens chacun des processeurs est autonome puisqu'il n'a besoin que de la connaissance des états précédents des nœuds voisins. La nature dynamique des traitements entraîne que $V_{i0}^{(t+1)}$ peut être différent de $V_{i0}^{(t)}$. On aboutit alors à un modèle parallèle et asynchrone (MIMD)

Discussion générale:
Les opérateurs de bas niveau sont destinés à traiter un flot énorme d'informations et les besoins en capacités de calculs se trouvent élevés. La rapidité du traitement est alors cruciale. De plus le volume de communication interprocesseurs se trouve augmenté et par la suite l'optimisation de la synchronisation des échanges et le choix de la topologie d'interconnexions sont des facteurs de performance.
D'autre part en raison de la réduction de l'espace des données on peut penser que les opérateurs de bas niveau n'ont pas de contrainte aussi sévère que leurs prédécesseurs. Toutefois, il est nécessaire de tenir compte de la nature itérative des traitements (convergence du processus) et de l'indéterminisme correspondant.
De ce fait, nous nous sommes intéressés aux algorithmes du traitement d'images de bas niveau en ayant comme objectif de diminuer le temps total d'exécution. Nous avons exploité une autre forme de parallélisme qui nécessite un modèle asynchrone MIMD et qui prévoit un découpage en tâches (qui n'est pas toujours évident à déterminer) de l'algorithme séquentiel.

II. PRESENTATION DE LA METHODE DE PARALLELISATION

1. Introduction .

Un algorithme séquentiel est représenté par une séquence d'instructions. Notre objectif consiste alors à décomposer l'algorithme en une suite de tâches et permettre,

à l'aide d'une méthode de parallélisation, d'avoir l'exécution simultanée de plusieurs tâches.
En principe, il existe deux sortes de traitements parallèles: le parallélisme vrai et le pipelining. Le parallélisme vrai nécessite que les traitements (tâches) soient indépendants
les uns des autres pour pouvoir s'exécuter simultanément sur des unités distinctes, alors que pour le pipelining, l'exécution complète d'un traitement passe par plusieurs étapes consécutives, chaque étape étant prise par une unité distincte, l'ensemble de ces unités forme un pipeline.

2. Traitements parallèles.
Le parallélisme vrai et le pipelining peuvent coexister et être mis en œuvre à différents niveaux : opération, instruction, application. Le parallélisme des opérations et des instructions, qui est interne à un processeur, est qualifié de bas niveau alors que la coopération de plusieurs processeurs à l'exécution d'une application correspond à un parallélisme de haut niveau.
Le parallélisme de haut niveau:
Le parallélisme de haut niveau est difficile à exploiter. En effet, pour utiliser ce type de parallélisme, on considère qu'une application est décomposée en tâches. Cette décomposition fait apparaître des relations de dépendances entre les tâches qui sont dues aux transferts des données. Une tâche reçoit des données de ses prédécesseurs, effectue des calculs sur ces données et envoie les résultats en sortie vers ses successeurs. Les relations de précédence traduisent le parallélisme interne de l'application: en effet, il existe entre les tâches des conditions d'exécutabilité car certaines actions ne peuvent commencer avant que d'autres ne soient terminées, et des conditions de validité car il faut préserver la cohérence des informations transmises ou mémorisées. Ces deux contraintes se traduisent par des relations de précédence qui dépendent de la logique des tâches à accomplir et qui fixent leur déroulement dans le temps (ordonnancement des tâches).
3. Concepts de base.
Il existe deux types de dépendances de programmes [Bacon +94; Binkley +95]: Dépendance de données et dépendance de contrôle.
Exemples :
a) *i1 : if x < 20 then*
i2 : x :=x+1
Ces deux instructions ont une dépendance de contrôle car le prédicat *i1* détermine si *i2* sera exécutée.
Deux instructions ont une dépendance de données si elles ne peuvent pas être exécutées simultanément à cause d'un conflit d'utilisation de la même variable. On en distingue trois types :
b.1)
x:=1 *i3*
*y :=z * x* *i4*
i3 et i4 ont une dépendance de flux car i3 sera exécutée en premier et i3 définit une valeur qui est utilisée par i4
b.2) *y := x/20* *i5*
x :=z+4 *i6*
i5 et i6 ont une anti-dépendance car i5 utilise une valeur d'une variable x et i6 définit une nouvelle valeur de x (i5 doit être exécutée avant i6).

b.3)
```
while x<=10 do      i7
y :=y+2/x           i8
x :=x *10           i9
ad
```

Dans ce cas il y a une dépendance d'itération qui est aussi une dépendance de flux entre i8 et i9 qui sont des instructions d'itérations différentes. Il y a 2 cas particuliers de cette dépendance :

- La même instruction impliquée dans 2 itérations différentes.
- Une des instructions peut être l'instruction de contrôle.

La notion de relations de dépendances a été, initialement utilisée dans la phase d'optimisation dans le domaine de la compilation. Certains chercheurs proposent un graphe de dépendance de programmes qui représente des dépendances entre variables [JR 94] alors que d'autres proposent celui qui représente des dépendances entre opérations [Griswold 93+, Weise+94]. Pour notre part, nous avons choisi le modèle de dépendance [Ghoul 95] afin de modéliser nos algorithmes.

4. Le modèle de dépendance [Ghoul 95]

Un programme est modélisé sous forme de dépendances entre agents (variables).

- Tout agent, qui est dans ce cas une variable simple, est encapsulé par un ensemble d'activités qui sont :

  La déclaration (Dec); l'initialisation (New); l'assignation ou définition (Ass) L'utilisation (Val) et l'activité vide (∅) .

  Une activité est définie par :<N. Act. Num> où N est le nom de l'agent, Act est le nom de l'activité et Num est le numéro de la dépendance où l'activité intervient.
- Un agent peut être prédéfini comme : Integer, Real, Boolean.
- Etant donné que le flux de contrôle est défini entre un prédicat et un ensemble d'instructions, l'agent de contrôle ε est introduit pour représenter ce prédicat.
- L'activité ∅ est introduite pour représenter l'anti-dépendance. Il y a 3 formes d'anti-dépendances :

  entre (1) la déclaration et la première assignation d'une variable (2) une utilisation et une assignation de la même variable (3) une assignation d'une variable v1 et une instruction de contrôle c1 où v1 est utilisée dans une instruction du corps de c1.

La modélisation des relations entre agents revient à la modélisation des dépendances entre leurs activités. Les différentes relations existantes au sein d'une même instruction ou entre instructions différentes d'un programme sont représentées par un ensemble de dépendances entre activités d'agents

La relation entre les activités au sein d'une même instruction définit une dépendance locale. Ces dépendances locales peuvent avoir des relations entre elles induites par des dépendances entre activités d'instructions différentes ou d'itérations différentes, ce qui définit l'ensemble des dépendances externes.

Toutes ces dépendances sont représentées explicitement dans un même graphe GDAA (Graphe de Dépendances entre Activités d'Agents) par un ensemble d'arcs entre activités d'agents sous contraintes : $Ai \xrightarrow{ctr} Ac$

Une contrainte de dépendance dénote le type de cette dernière. Elle est composée d'un triplet : **< condition, expression, sens >** . La condition doit avoir la valeur vraie pour que l'activité correspondante peut s'exécuter. L'expression est la valeur calculée pour

cette activité, alors que le sens définit la nature de la dépendance (itération, anti-dépendance, …).

4.1. Notion d'arbre post-dominateur :
Il représente explicitement l'enchaînement des instructions d'un programme. Pour le construire, on génère une racine systématique " Entrée " et l'on associe à chaque instruction i un nœud (i, cd) où cd est, soit ε si cette instruction dépend d'une instruction de contrôle, soit ∅ dans le cas contraire. On définit un arc orienté (1) du nœud entrée vers tout nœud de type (i, ∅) (2) du nœud (i, cd) d'une instruction de contrôle vers tout nœud de type (j, ε)

4.2 Définition des dépendances locales:
A partir du programme et l'arbre post-dominateur correspondant on construit les éléments de la relation interne suivant le type d'instruction:

a) Pour chaque instruction i de déclaration d'une variable x de type T on a:

$$(x.dec.i) < \varnothing, t, \varnothing) > \{< T, New, i >\}$$

b) Pour chaque instruction i d'affectation de la variable *x* qui utilise une ou plusieurs variables y_k , on aura:

$$(x.Ass.i) < cd, exp_i, sens > \{(y_1.Val.i)...(y_k.Val.i),[\varepsilon.Val.i][Z.\varnothing.i]\}$$

Si l'instruction i ε au corps de l'instruction de contrôle j

Dans le cas où il y a une anti-dépendance entre l'instruction i définissant x et l'instruction j définissant Z

où
cd = ∅ ou ε d'après l'arbre post-dominateur
exp_i = partie droite de l'affectation n° i
sens : ← : anti-dépendance
∅ : flux ou vide

c) Pour chaque instruction de contrôle qui utilise une ou plusieurs variables y_k.

$$(\varepsilon.Ass.i) < cd, expr, sens > \{(y_1.Val.i)...(y_k.Val.i),[\varepsilon.Val.i][Z.\varnothing_{\varnothing}.i]\}$$

où sens : ∅ : condition * : itération

4.3 Définition des dépendances externes:
La relation externe relie sous contrainte une activité (d'utilisation) à une activité (de définition) du même agent dans des instructions différentes telles qu'on a :

(x. val. j) < contrainte > (x. ass. i)

où j représente le numéro dans le programme de la dépendance interne correspondante à l'instruction qui utilise la variable x. Les numéros i et j peuvent être identiques dans le cas d'une instruction appartenant à une boucle. Si x est un prédicat de contrôle ε la correspondance entre les numéros i et j est donnée par l'arbre post-dominateur (spécifiée par l'arc du nœud (i, cd) au nœud (j, ε)) sinon c'est par programme.
D'autre part pour chaque instruction de déclaration i d'une variable x de type T, la dépendance externe concerne la relation entre l'activité d'utilisation x. ∅.j et

l'activité x dec.i où j représente le numéro de la dépendance interne correspondant à la première affectation de la variable x.

4.4 Construction de GDAA (Graphe de dépendance entre les activités d'agents)
A partir du programme, on construit le GDAA en définissant les deux ensembles des relations de dépendance locales du et externes ud.
Chaque instruction i du programme engendre une dépendance locale de numéro i entre ses activités et chaque relation entre instructions différentes (ou même instruction appartenant à des itérations différentes) définit une dépendance externe.

5. Mise en œuvre de la parallélisation.
5.1 Caractéristiques des algorithmes.
L'étude et l'analyse des algorithmes séquentiels considérés, nous permet de constater et de faire la décomposition fonctionnelle suivante:

- Tout algorithme (tâche globale) est décrit par une suite de tâches.
- Toute tâche est caractérisée par un ensemble de tâches élémentaires qui forme un bloc d'itération dont le point d'entrée est une boucle simple ou composée (si l'image est respectivement sous forme d'un vecteur ou une matrice à deux dimensions)
- Toute tâche a une entrée unique et une sortie unique.
- Toute tâche élémentaire est caractérisée par une certaine transformation (fonction) sur un pixel x (i) (ou x (i, j)).
- La tâche élémentaire est la tâche qui formera un tout en elle même et ne sera plus décomposée.

5.2 Dépendance des tâches.
Nous avons défini des relations de dépendances et d'indépendances entre les tâches. Ces définitions sont liées à l'indépendance d'exécution.

a) Indépendance d'exécution:
Une exécution d'une tâche T1 est indépendante de l'exécution d'une tâche T2 si seulement si T1 et T2 peuvent s'exécuter simultanément et produire des résultats identiques à ceux produits par l'exécution de T1 suivie de celle de T2.

b) Indépendance de données:
Deux tâches T1 et T2 ont une indépendance de données si seulement si la $i^{è}$ exécution de T1 est indépendante de la $i^{è}$ exécution de T2.
(Nous dirons encore que les exécutions sont sans conflit)

c) Dépendance vraie de données entre tâches:
On dit qu' une tâche T2 est en dépendance vraie d' une tâche T1 si seulement si la $i^{è}$ exécution de T1 ne nécessite pas la $i^{è}$ exécution de T2 et la $i^{è}$ exécution de T2 nécessite la $i^{è}$ exécution de T1.
(Nous dirons que T1 doit précéder T2)

d) Antidépendance de données entre tâches:
Deux tâches T1 et T2 sont dites antidépendantes si seulement si la $(i+1)^{è}$ exécution de T2 nécessite la $i^{è}$ exécution de T1.

5.3 Analyse des dépendances et extraction du parallélisme.

- Vu que les algorithmes présentent un niveau d'abstraction élevé par rapport aux codes programmes, on adopte les restrictions suivantes:
  - Toute variable identifiée dans l'algorithme sera encapsulée par les activités: Ass; Val; ∅.
  - Les agents prédéfinis comme (Integer, Boolean, …) n'existent pas.

- Les variables n'intervenant pas directement dans l'opération de calcul d'une tâche ne seront pas considérées.
- Les variables identifiées dans les algorithmes sont généralement des variables indicées; nous considérons alors chaque composante comme une variable différente. Néanmoins ces variables présentent une relation due à l'indice (car c'est la valeur de ce dernier qui identifie chaque composante). Par conséquent, on enrichie le sens de la contrainte par un nouveau type de dépendance temporelle: en effet si x (i) désigne la variable calculé au temps t alors la variable x (i-2) dépendra de la définition de x (i) au temps t-2.
  D'autre part, vu que le temps t (caractérisé par un flux continu) où la variable x (i) est calculé, ne peut être déterminé; on définit alors une variable $\mathbf{t_k}$ (**appelée distance de dépendance)** comme étant le nombre de pas nécessaire pour calculer une tâche élémentaire, tel que:
  1) Si l'image se présente sous forme d'un vecteur de dimension m
  $t_k = t_0 + (k - 1)\, m + i - 1$
  Où k = numéro de la tâche
  i = le $i^{è}$ pixel traité
  t_0 = le pas initial du point d'entrée de la $1^{ère}$ tâche.
  Dans le cas où la boucle du bloc caractérisant une tâche est décrémentale alors $t_k = t_0 + (k - 1)\, m - i$
  2) Si l'image est à 2 dimensions (m, m)
  $t_k = t_0 + (k - 1)\, m^2 + (i - 1)\, m + j - 1$
- Dans le cas où une instruction définit un calcul récursif sur une variable x (i) alors (x (i). Val . *l*) < contrainte > (x (j).Ass.*l*) définit une relation externe dans la même instruction *l* entre une activité d'utilisation d'une variable x (i) à une activité de définition d'une autre variable x (j).

⇒ Enfin pour déterminer quelles sont les tâches qui peuvent s'exécuter en parallèle, on procède comme suit:

a- A partir du GDAA, on déduit un graphe de précédence où les tâches élémentaires correspondent aux sommets et les arcs, qui matérialisent les relations, sont annotés par les contraintes qui déterminent le type de dépendance.

b- la détection des dépendances ne s'effectue qu'entre un couple de tâches élémentaires
$T_{k1,i}$ et $T_{k2,i}$ qui sont reliées par un arc orienté.

c- Pour extraire le parallélisme, on définit les conditions suivantes:

1. **Si** deux tâches élémentaires $T_{k1,i}$ et $T_{k2,i}$ admettent une indépendance de données (∀ l'exécution).
   Alors, les tâches T_{k1} et T_{k2} peuvent s'exécuter en parallèle.
2. **Si** deux tâches élémentaires $T_{k1,i}$ et $T_{k2,i}$ ont une dépendance vraie de données ou une antidépendance de données.
   Alors, **Si** la différence entre les pas respectifs t_{k1} et t_{k2} est un facteur de m (ie $t_{k1} - t_{k2} = fm$
   (ou fm^2 si l'image est à deux dimensions), f est un facteur qui varie de 0 à N (N : nombre entier).
   Alors les tâches T_{k1} et T_{k2} peuvent s'exécuter en parallèle.

6. Application .

6.1. Introduction:

Nous nous sommes intéressés particulièrement aux algorithmes de détection de contours. Par la suite deux algorithmes représentatifs des deux techniques utilisées ont fait l'objet de notre méthode de parallélisation.

6.2. Parallélisation de l'algorithme de DERICHE :

6.2.1 Description :

L'algorithme de Deriche est un des algorithmes d'extraction de contour qui procèdent par optimisation de critères prenant en compte un modèle prédéfini du contour à détecter.

Ce problème d'optimisation a conduit Deriche [comme Canney] à trouver un filtre dont la réponse impulsionnelle infinie [finie pour Canney] *f(x)* est une solution d'une équation différentielle :

$$2 f(x) - 2 \lambda_1 f''(x) + 2 \lambda_2 f''''(x) + \lambda_3 = 0 \quad \ldots\ldots\ldots (1)$$

et vu que l'intervalle considéré est]-∞, +∞[et les conditions aux bornes sont

$f(0) = 0$; $f(\infty) = 0$; $f'(0) = s$; $f'(\infty) = 0$,

la solution de l'équation (1) est : $f(x) = c.x.e^{-\alpha|x|}$

Cet opérateur est très performant, simple et représente un seul paramètre α, ce paramètre est ajusté pour favoriser une bonne localisation ou un bon rapport signal bruit.

6.2.2 Implémentation récursive de l'algorithme en 1D :

Tout d'abord, le détecteur est appliqué à l'image puis le résultat est lissé par un filtre de lissage, le lissage est nécessaire vu que le gradient obtenu en sortie du 1er filtre est plus ou moins sensible (tout dépend du choix du paramètre α).
La réponse impulsionnelle *f(x)* du lissage est prise comme une intégrale de *f(x)*.

a) Mise en œuvre du filtre récursif *f(x)* :

Il est obtenu en appliquant la transformée en Z :

$F(Z) = \Sigma f(n) Z^{-1}$ et $f(n) = f^-(n) + f^+(n)$ car *f* est non causale

où $f^-(n) = \begin{cases} 0 & n > 0 \\ sne^{\alpha n} & n \le 0 \end{cases}$ et $f^+(n) = \begin{cases} sne^{-\alpha n} & n \ge 0 \\ 0 & n < 0 \end{cases}$

soit alors $Z^+(Z) = \dfrac{c_1 Z^{-1}}{1 + b_1 Z^{-1} + b_2 Z^{-2}}$ et $Z^-(Z) = \dfrac{-c_1 Z}{1 + b_1 Z + b_2 Z}$

où $c_1 = -(1 - e^{-\alpha})^2$ et $b_1 = -2e^{-\alpha}$ et $b_2 = e^{-2\alpha}$

F^+ et F^- correspondent à 2 fonctions de transfert de filtres récursifs stables (car leurs pôles ∈ à un cercle unité pour α réel positif).

Donc, soit $I(i)$ le signal d'entrée du système défini par les 2 fonctions précédentes,
la sortie $X(i)$ peut être obtenue de manière récursive par les équations suivantes :

$X^+(i) = I(i-1) + 2e^{-\alpha} X^+(i-1) - e^{-2\alpha} X^+(i-2)$; $i = \overrightarrow{1.M}$

$X^-(i) = I(i+1) + 2e^{-\alpha} X^-(i+1) - e^{-2\alpha} X^-(i+2)$; $i = \overrightarrow{M.1}$

$X(i) = c_1 \left[X^+(i) - X^-(i) \right]$ pour $i = \overrightarrow{1.M}$ et $c_1 = -(1 - e^{-\alpha})$

b) Mise en œuvre du filtre *h(x)* :

Le gradient obtenu peut être lissé par la primitive $h(n)$ de $f(n)$ où :

$h(x) = k(\alpha |x| + 1)e^{-\alpha|x|}$

et d'une manière similaire, en appliquant les techniques de la transformée en *Z*, on obtient

la sortie ci-dessous où $I(m)$ est le signal d'entrée du système défini par H^- et H^+ (fonctions de transfert des filtres récursifs) :

$$X^+(i) = KI(i) + Ke^{-\alpha}(\alpha-1)I(i-1) + 2e^{-\alpha}X^+(i-1) - e^{-2\alpha}X^+(i-2); i = \overrightarrow{1.M}$$

$$X^-(i) = Ke^{-\alpha}(\alpha+1)I(i+1) - e^{-2\alpha}I(i+2) + 2e^{-\alpha}X^-(i+1) - e^{-2\alpha}X^-(i+2); \; i = \overrightarrow{M.1}$$

$$X(i) = X^+(i) + X^-(i) \quad \text{pour} \quad i = \overrightarrow{1.M}$$

6.2.3 Modélisation de l'algorithme de DERICHE par le graphe de dépendance :

a) L'algorithme :

$i = 1$ (1)

tant que $i \le M$ faire (2) ε1

| $X^+(i) = I(i-1) + 2e^{-\alpha}X^+(i-1) - e^{-2\alpha}X^+(i-2)$ (3)
| $i = i + 1$ (4)

fin

$i = M$ (5)

tant que $i \ge 1$ faire (6) ε2

| $X^-(i) = I(i+1) + 2e^{-\alpha}X^-(i+1) - e^{-2\alpha}X^-(i+2)$ (7)
| $i = i - 1$ (8)

fin

$i = 1$ (9)

tant que $i \le M$ faire (10) ε3

| $X(i) = -1.(1-e^{-\alpha}).\left[X^+(i) - X^-(i)\right]$ (11)
| $i = i + 1$ (12)

fin

$i = 1$ (13)

tant que $i \le M$ faire (14) ε4

| $Y^+(i) = KX(i) + Ke^{-\alpha}X^+(\alpha-1)X(i-1) + 2e^{-\alpha}Y^+(i-1) - e^{-2\alpha}Y^+(i-2)$ (15)
| $i = i + 1$ (16)

fin

$i = M$ (17)

tant que $i \ge 1$ faire (18) ε5

| $Y^-(i) = Ke^{-\alpha}(\alpha+1)X(i+1) - e^{-2\alpha}X(i+2) + 2e^{-\alpha}Y^-(i+1) - e^{-2\alpha}Y^-(i+2)$ (19)
| $i = i - 1$ (20)

fin

$i = 1$ (21)

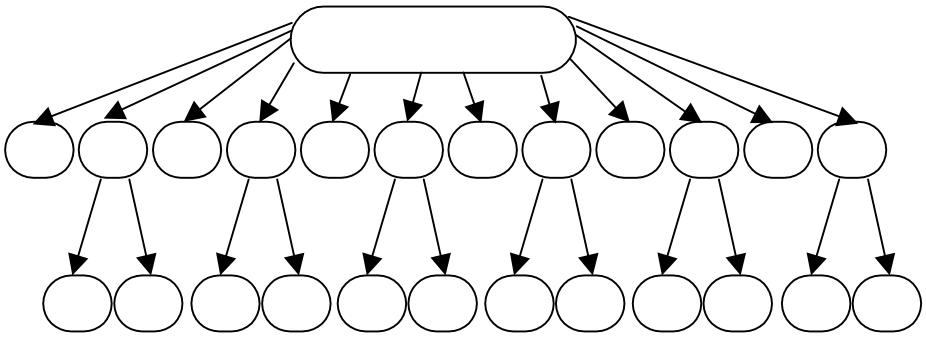
tant que $i \leq M$ faire (22) ε6

$Y(i) = Y^{+}(i) + Y^{-}(i)$ (23)

$i = i + 1$ (24)

fin

b) Construction de l'arbre post-dominateur :



c) Identification des tâches élémentaires et variables :

- Tâches élémentaires :

$T_{1,i} : X^{+}(i) = 1(i-1) + 2e^{-\alpha} X^{+}(i-1) - e^{-2\alpha} X^{+}(i-2)$

$T_{2,i} : X^{-}(i) = 1(i+1) + 2e^{-\alpha} X^{-}(i+1) - e^{-2\alpha} X^{-}(i+2)$

$T3_{,i} : X(i) = -1(1 - e^{-\alpha}).[X^{+}(i) - X^{-}(i)]$

$T_{4,i} : Y^{+}(i) = KX(i) + Ke^{-\alpha}(\alpha - 1)X(i-1) + 2e^{-\alpha} Y^{+}(i-1) - e^{-2\alpha} Y^{+}(i-2)$

$T_{5,i} : Y^{-}(i) = Ke^{-\alpha}(\alpha+1)X(i+1) - e^{-2\alpha} X(i+2) + 2e^{-\alpha} Y^{-}(i+1) - e^{-2\alpha} Y-(i+2)$

$T_{6,i} : Y(i) = Y^{+}(i) + Y^{-}(i)$

- Variables :

$X^{+}(i) ; X^{+}(i-1) ; X^{+}(i-2) ; X^{-}(i+1) ; X^{-}(i+2) ; X^{-}(i)$

$X(i) ; Y^{+}(i) ; X(i-1) ; Y^{+}(i-1) ; Y^{+}(i-2) ; Y^{-}(i)$

$X(i+1) ; X(i+2) ; Y^{-}(i+1) ; Y^{-}(i+2) ; Y(i)$

- A chaque tâche élémentaire $T_{k,i}$ correspond un pas t_k nécessaire pour son calcul et $\sum_{k=1}^{6} t_k$ correspond au nombre de pas nécessaires pour calculer séquentiellement les $T_{k,i}$ ($k = 1 \ldots 6$).

d) Détermination des dépendances locales :

(1) $(X^{+}(i).ass.3) < \varepsilon 1, exp_3, t_1 > \{(X^{+}(i-1).val.3)\ (X^{+}(i-2).val.3)\ (\varepsilon 1.val.3)\}$

(2) $(X^{-}(i).ass.7) < \varepsilon 2, exp_7, t_2 > \{(X^{-}(i+1).val.7)\ (X^{-}(i+2).val.7)\ (\varepsilon 2.val.7)\}$

(3) $(X(i).ass.11) < \varepsilon 3, exp_{11}, t_3 > \{(X^{+}(i).val.11)\ (X^{-}(i).val.11)\ (\varepsilon 3.val.11)\}$

(4) $(Y^{+}(i).ass.15) < \varepsilon 4, exp_{15}, t_4 > \{(X(i).val.15)\ (X(i-1).val.15)(Y^{+}(i-1).val.15)$

$(Y^{+}(i-2).val.15)\ (\varepsilon 4.val.15)\}$

(5) $(Y^{-}(i).ass.19) < \varepsilon 5, exp_{19}, t_5 > \{(X(i+1).val.19)\ (X(i+2).val.19)(Y^{-}(i+1).val.19)$

$(Y^{-}(i+2).val.19)\ (\varepsilon 5.val.19)\}$

(6) $(Y(i).ass.23) < \varepsilon 6, exp_{23}, t_6 > \{(Y^{+}(i).val.23)\ (Y^{-}(i).val.23)\ (\varepsilon 6.val.23)\}$

e) Détermination des dépendances externes :

(7) $(X^{+}(i-1).val.3) < \Phi, X^{+}(i-1), t_1 - 1 > (X^{+}(i).ass.3)$

(8) $(X^{+}(i-2).val.3) < \Phi, X^{+}(i-2), t_1 - 2 > (X^{+}(i).ass.3)$

(9) $(X^{-}(i+1).val.7) < \Phi, X^{-}(i+1), t_2 + 1 > (X^{-}(i).ass.7)$

(10) $(X^{-}(i+2).val.7) < \Phi, X^{-}(i+2), t_2 + 2 > (X^{-}(i).ass.7)$

(11) $(X^{+}(i).val.11) < \Phi, X^{+}(i), t_1 > (X^{+}(i).ass.3)$

(12) $(X^{-}(i).val.11) < \Phi, X^{-}(i), t_2 > (X^{-}(i).ass.7)$

(13) $(X(i-1).val.15) < \Phi, X(i-1), t_3 - 1 > (X(i).ass.11)$

(14) $(X(i).val.15) < \Phi, X(i), t_3 > (X(i).ass.11)$

(15) $(Y^{+}(i-1).val.15) < \Phi, Y^{+}(i-1), t_4 - 1 > (Y^{+}(i).ass.15)$

(16) $(Y^{+}(i-2).val.15) < \Phi, Y^{+}(i-2), t_4 - 2 > (Y^{+}(i).ass.15)$

(17) $(X(i+1).val.19) < \Phi, X(i+1), t_3 + 1 > (X(i).ass.11)$

(18) $(X(i+2).val.19) < \Phi, X(i+2), t_3 + 2 > (X(i).ass.11)$

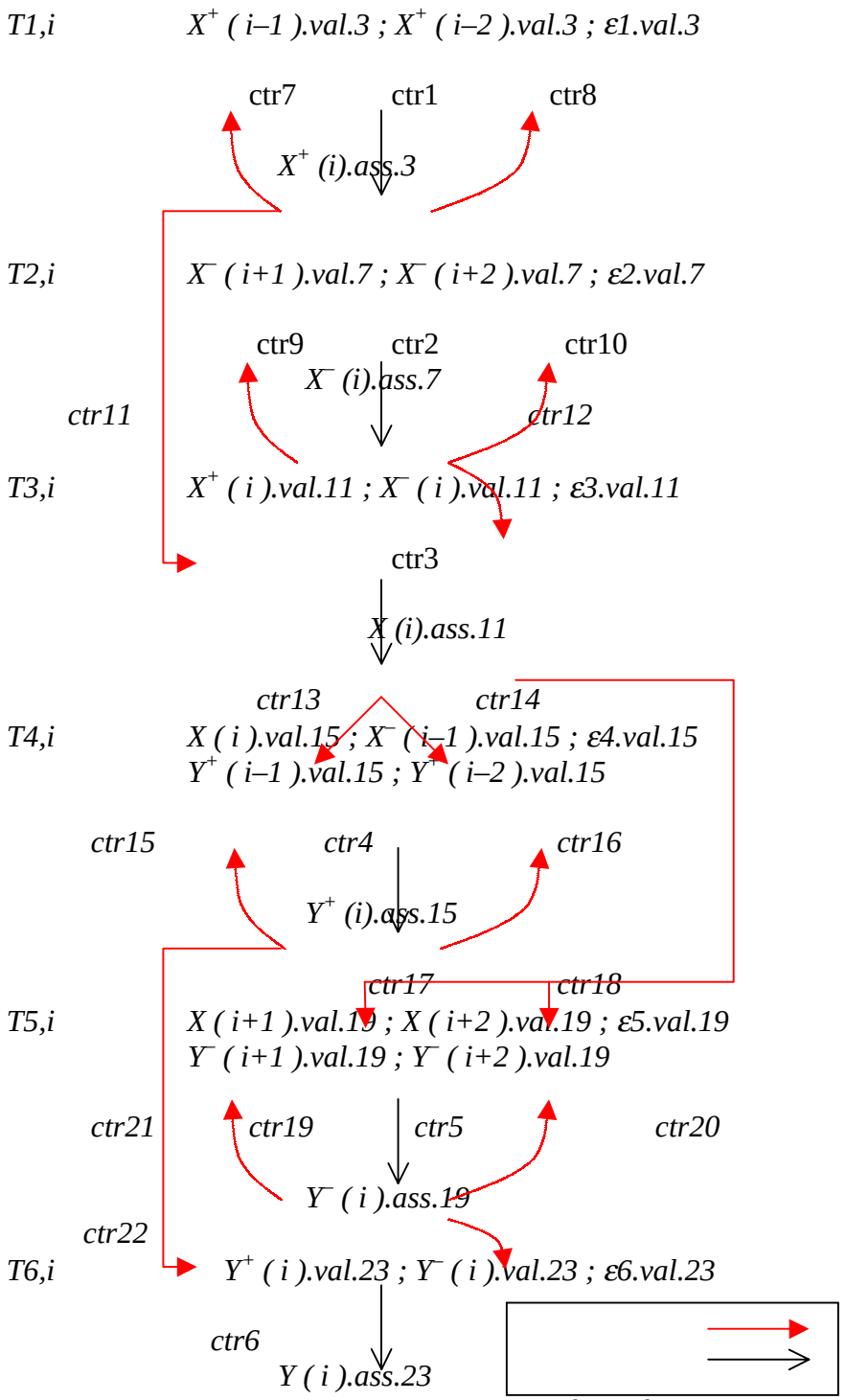
(19) $(Y^{-}(i+1).val.19) < \Phi, Y^{-}(i+1), t_5 + 1 > (Y^{-}(i).ass.19)$

(20) $(Y^{-}(i+2).val.19) < \Phi, Y^{-}(i+2), t_5 + 2 > (Y^{-}(i).ass.19)$

(21) $(Y^{+}(i).val.23) < \Phi, Y^{+}(i), t_4 > (Y^{+}(i).ass.15)$

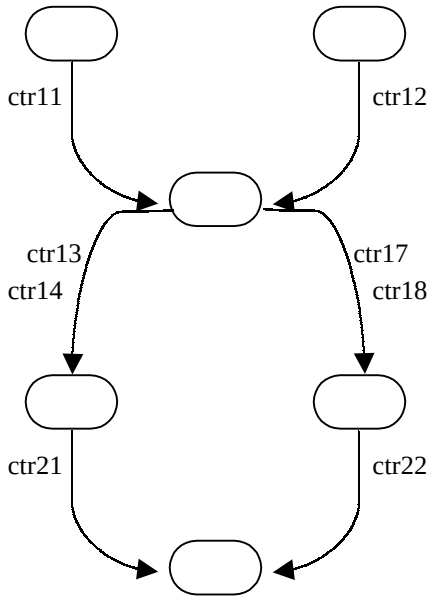
(22) $(Y^{-}(i).val.23) < \Phi, Y^{-}(i), t_5 > (Y^{-}(i).ass.19)$

f) Construction du GDAA:



NB : « ctr n » est la contrainte qui correspond à la dépendance locales ou externes numéro "n".

g) Détermination du graphe de précédence des tâches élémentaires :



h) Détermination du graphe de l'algorithme parallèlisé :

- $T_{1,i}$ et $T_{2,i}$ ainsi que $T_{4,i}$ et $T_{5,i}$ ont une indépendance de données, par conséquent T_1 et T_2 ainsi que T_4 et T_5 sont des tâches parallélisables :

  $T_1 // T_2$ Ainsi que $T_4 // T_5$

- $T_{1,i}$ et $T_{3,i}$ ont une dépendance vraie de données; d'autre part :

  $t_3 - t_1 = (t_0 + 2m + i - 1) - (t_0 + 0m + i - 1) = 2m \Rightarrow f = 2$.

  Donc $T_1 // T_3$

- $T_{3,i}$ et $T_{2,i}$ ont une dépendance vraie de données ; d'autres part :

  $t_3 - t_2 = (t_0 + 2m + i - 1) - (t_0 + m + i) = 2m + 2i - 1 \neq f\,m$.

  Donc T_3 reste séquentielle avec T_2

- $T_{3,i}$ et $T_{4,i}$ ont une dépendance vraie et une anti-dépendance de données, car suivant les contraintes ctr13 et ctr14, $T_{4,i}$ dépend de $T_{3,i}$ de la $(i-1)^{ème}$ et la $i^{ème}$ exécutions.

  $t_4 - t_3 = (t_0 + 3m + i - 1) - (t_0 + 2m + i - 1) = m = f\,m$ où $f = 1$

  Donc $T_3 // T_4$

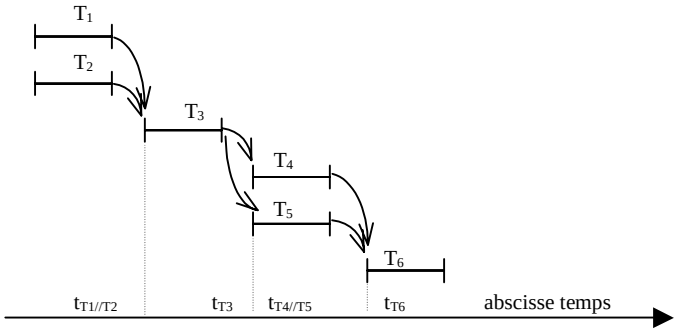
- $T_{5,i}$ et $T_{3,i}$ n'ont ni anti-dépendance, ni dépendance vraie de données car suivant les contraintes ctr17 et ctr18, $T_{5,i}$ dépend de $T_{3,i}$ de la $(i + 1)^{ème}$ et $(i + 2)^{ème}$ exécution

  T_5 reste séquentielle avec T_3

- $T_{6,i}$ et $T_{4,i}$ ont une dépendance vraie de données ; d'autre part
  $t_6 – t_4 = t_0 + 5m + (i – 1) – (t_0 + 3m + i - 1) = 2m$
  $\Rightarrow$ T_4 // T_6

- $T_{6,i}$ et $T_{5,i}$ ont une dépendance vraie de données et
  $t_6 – t_5 = t_0 + 5m + (i – 1) – (t_0 + 4m - i) = m + 2i - 1 \neq f$ m.
  Donc T_6 est séquentielle à T_5

Finalement le graphe ou le schéma de communication des tâches de l'algorithme parallélisé sera comme suit :



6.3. Parallélisation de l'algorithme utilisant les opérateurs locaux de dérivation:

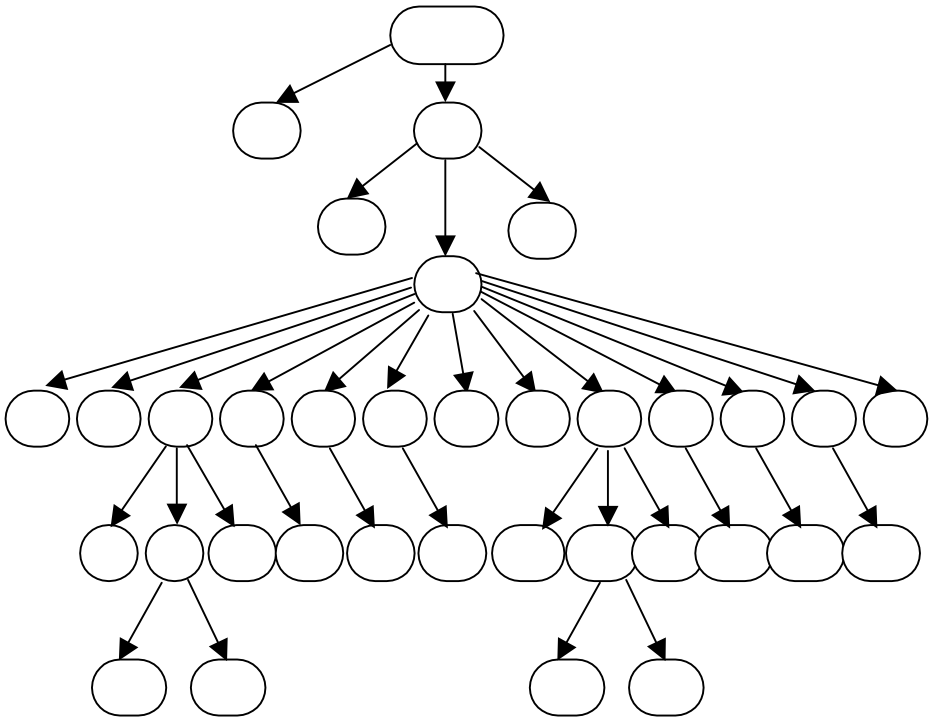
6.3.1. Description

Cet algorithme réalise la détection de contours suivant 8 directions :

3 2 1
4 × 0
5 6 7

En faisant 8 convolutions séquentielles entre une image entrée et les masques correspondants.
La même fonction est appliquée pour les 8 convolutions, pour cela on étudie le cas de 2 convolutions successives et la conclusion sera généralisée par la suite aux 8 étapes de l'algorithme (figure 1) :

6.3.2. Construction de l'arbre post-dominateur :



6.3.3 Identification des tâches élémentaires :

$T_{1,i,j}$: sum = $\sum_{-1}^{1}\sum_{-1}^{1} \text{mask1} * \text{image}(i, j)$

Si sum > 255 Alors sum = 255

Si sum < 0 Alors sum = 0

Si sum > x (i, j) Alors x (i, j) = sum

$T_{2,i,j}$: sum = $\sum_{-1}^{1}\sum_{-1}^{1} \text{mask2} * \text{image}(i, j)$

Si sum > 255 Alors sum = 255

Si sum < 0 Alors sum = 0

Si sum > x (i, j) Alors x (i, j) = sum

Identification des variables :

x (i, j) ; ε5 ; ε6 ; ε7 ; ε10 ; ε11 ; ε12. sum.

6.3.4 Détermination des dépendances locales :

1- $(sum.ass.5) < \varepsilon_2, \exp_5, t_1 > \{(\varepsilon_2.val.5)\}$

2- $(sum.ass.10) < \varepsilon_2, \exp_{10}, t_1 > \{(\varepsilon_2.val.10);(sum.val.10)\}$

3- $(\varepsilon_5.ass.13) < \varepsilon_2, pred_{13}, t_1 > \{(\varepsilon_2.val.13);(sum.val.13)\}$

4- $(sum.ass.14) < \varepsilon_5, \exp_{14}, t_1 > \{(\varepsilon_5.val.14)\}$

5- $(\varepsilon_6.ass.15) < \varepsilon_2, pred_{15}, t_1 > \{(\varepsilon_2.val.15);(sum.val.15)\}$

6- $(sum.ass.16) < \varepsilon_6, \exp_{16}, t_1 > \{(\varepsilon_6.val.16)\}$

7- $(\varepsilon_7.ass.17) < \varepsilon_2, pred_{17}, t_1 > \{(\varepsilon_2.val.17);(sum.val.17);(x(i,j).val.17)\}$

8- $(x(i,j).ass.18) < \varepsilon_7, \exp_{18}, t_1 > \{(\varepsilon_7.val.18);(sum.val.18)\}$

9- $(sum.ass.19) < \varepsilon_8, \exp_{19}, t_1 > \{(\varepsilon_2.val.19)\}$

10- $(sum.ass.24) < \varepsilon_9, \exp_{24}, t_1 > \{(\varepsilon_9.val.24);(sum.val.24)\}$

11- $(\varepsilon_{10}.ass.27) < \varepsilon_2, pred_{27}, t_1 > \{(\varepsilon_2.val.27);(sum.val.27)\}$

12- $(sum.ass.28) < \varepsilon_{10}, \exp_{28}, t_1 > \{(\varepsilon_{10}.val.28)\}$

13- $(\varepsilon_{11}.ass.29) < \varepsilon_2, pred_{29}, t_1 > \{(\varepsilon_2.val.29);(sum.val.29)\}$

14- $(sum.ass.30) < \varepsilon_{11}, \exp_{30}, t_1 > \{(\varepsilon_{11}.val.30)\}$

15- $(\varepsilon_{12}.ass.31) < \varepsilon_2, pred_{31}, t_1 > \{(\varepsilon_2.val.31);(sum.val.31);(x(i,j).val.31)\}$

16- $(x(i,j).ass.32) < \varepsilon_{12}, \exp_{32}, t_1 > \{(\varepsilon_{12}.val.32);(sum.val.32)\}$

6.3.5. Détermination des dépendances externes :

17- $(sum.val.10) < \Phi, som, t_1 > \{(sum.ass.5)\}$

18- $(sum.val.13) < \Phi, som, t_1 > \{(sum.ass.5);(sum.ass.10)\}$

19- $(sum.val.15) < \Phi, som, t_1 > \{(sum.ass.10);(sum.ass.5);(sum.ass.14)\}$

20- $(sum.val.17) < \Phi, som, t_1 > \{(sum.ass.5);(sum.ass.10);(sum.ass.14);(sum.ass.16)\}$

21- $(sum.val.18) < \Phi, som, t_1 > \{(sum.ass.5);(sum.ass.10);(sum.ass.14);(sum.ass.16)\}$

A partir du graphe post-dominateur, on détermine :

22- $(\varepsilon_5.val.14) < \Phi, \varepsilon_5, t_1 > \{(\varepsilon_5.ass.13)\}$

23- $(\varepsilon_6.val.16) < \Phi, \varepsilon_6, t_1 > \{(\varepsilon_6.ass.15)\}$

24- $(\varepsilon_7.val.18) < \Phi, \varepsilon_7, t_1 > \{(\varepsilon_7.ass.17)\}$

25- $(sum.val24) < \Phi, sum, t_1 > \{(sum.ass5);(sum.ass10);(sum.ass14);(sum.ass16);(sum.ass19)\}$

26- $(sum.val27) < \Phi, sum, t_1 > \{(sum.ass5);(sum.ass10);(sum.ass14);(sum.ass16);(sum.ass19);(sum.ass24)\}$

27- $(sum.val29) < \Phi, sum, t_1 > \{(sum.ass5);(sum.ass10);(sum.ass14);(sum.ass16);(sum.ass19);(sum.ass24);(sum.ass28)\}$

28- $(sum.val31) < \Phi, sum, t_1 > \{(sum.ass5);(sum.ass10);(sum.ass14);(sum.ass16);(sum.ass19);(sum.ass24);(sum.ass28);(sum.ass30)\}$

29- $(sum.val32) < \Phi, sum, t_1 > \{(sum.ass5);(sum.ass10);(sum.ass14);(sum.ass16);(sum.ass19);(sum.ass24);(sum.ass28);(sum.ass30)\}$

29'- $(x(i,j).val31) < \Phi, sum, t1 > (x(i,j).ass18)$

A partir du graphe post-dominateur, on détermine :

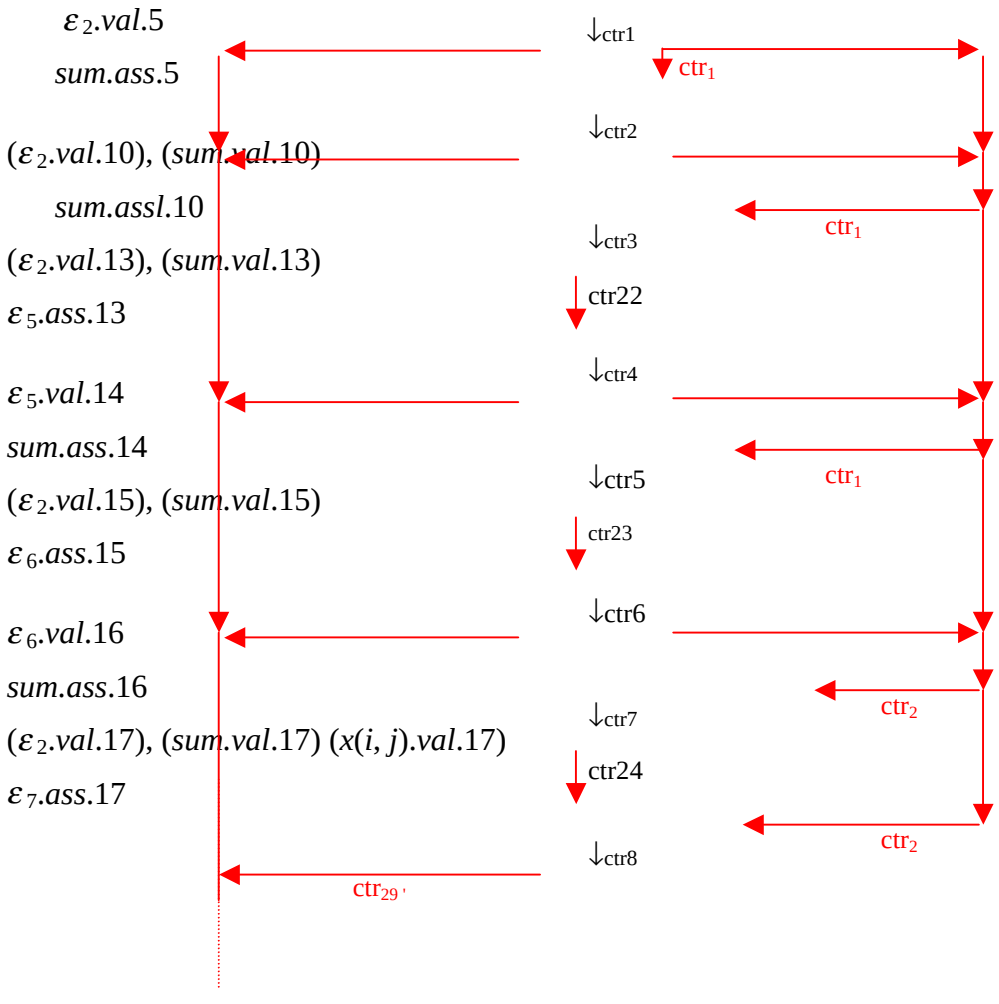
30- $(\varepsilon_{10}.val.28) < \Phi, \varepsilon_{10}, t_1 > \{(\varepsilon_{10}.ass.27)\}$

31- $(\varepsilon_{11}.val.30) < \Phi, \varepsilon_{11}, t_1 > \{(\varepsilon_{11}.ass.29)\}$

32- $(\varepsilon_{12}.val.32) < \Phi, \varepsilon_{12}, t_1 > \{(\varepsilon_{12}.ass.31)\}$

6.3.6. Construction du GDAA :

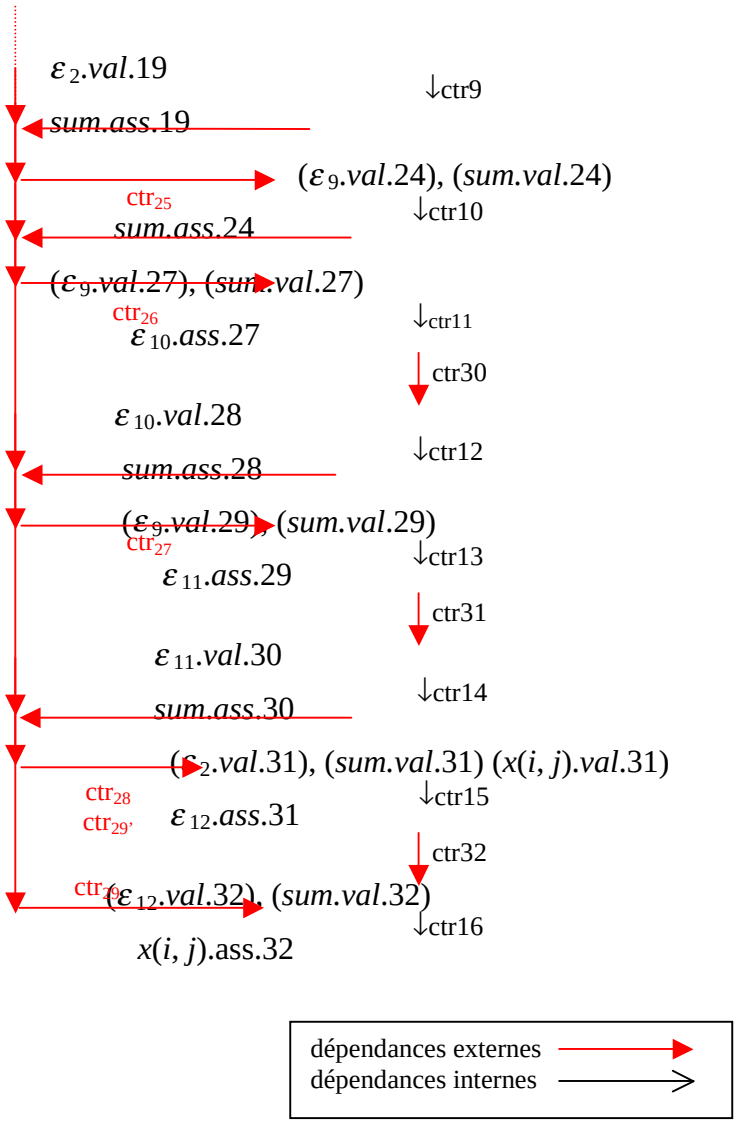
$T_{1,i,j}$:



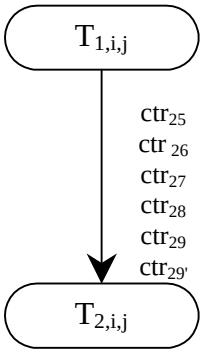
(ε_2.*val*.18), (*sum.val*.18)

x(*i*, *j*).*ass*.18

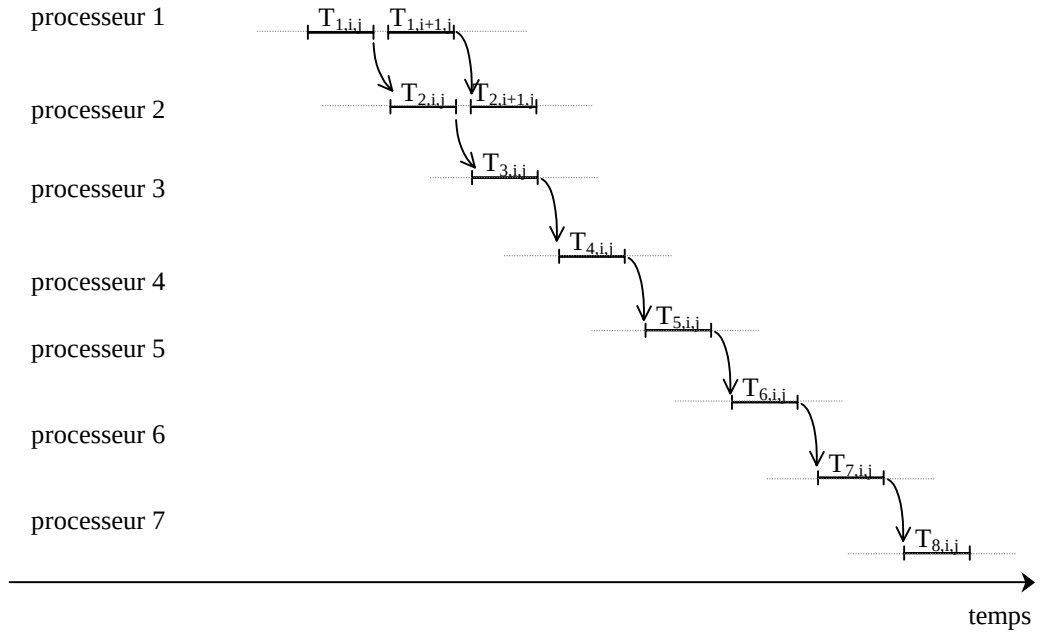
$T_{2,i,j}$:



Le graphe de précédence des tâches élémentaires est :



D’après les contraintes ctr_{25}, ctr_{26}, ctr_{27}, ctr_{28}, ctr_{29}, $ctr_{29'}$ les tâches élémentaires $T_{1,i,j}$ et $T_{2,i,j}$ ont une dépendance vraie de données ; de plus $t_1 - t_1 = 0$: ($f = 0$) car les 2 tâches élémentaires appartiennent au même bloc correspondant à la tâche $T_{i,j}$.
Par conséquent la tâche $T_{i,j}$ est parallélisable et le schéma de communication des 8 tâches élémentaires se présente comme suit :



```
Début :i = 1   (1)
Tant que i < M – 1 Faire        (2)      ε1
    j = 1                  (3)
    Tant que j < M – 1 Faire (4)      ε2
        sum = 0            (5)
        a = -1             (6)
        Tant que a < 2 Faire   (7)      ε3
            b = –1             (8)
            Tant que b < 2 Faire     (9)      ε4
                sum = sum + image [i + a][j + b] * mask1 [a + 1][b + 1] (10)
                b = b + 1    (11)
            Fin
                    a = a + 1          (12)
        Fin
        Si sum > 255 Alors    (13)     ε5
            sum = 255      (14)
        Fin
        Si sum < 0 Alors                (15)     ε6
            sum = 0                    (16)
        Fin
        Si sum > X [ i ][ j ] Alors              (17)     ε7
            X [ i ][ j ] = sum                 (18)
        Fin
        sum = 0           (19)
        a = -1            (20)
        Tant que a < 2 Faire   (21)     ε8
            b = -1             (22)
            Tant que b < 2 Faire      (23)     ε9
                sum = sum + image [i + a][j + b] * mask2 [a + 1][b + 1]  (24)
                b = b + 1        (25)
            Fin
                a = a + 1          (26)
        Fin
        Si sum > 255 Alors    (27)     ε10
            sum = 255      (28)
        Fin
        Si sum < 0 Alors                (29)     ε11
            sum = 0                    (30)
        Fin
        Si sum > X [ i ][ j ] Alors        (31)  ε12
            X [ i ][ j ] = sum           (32)
        Fin
        j = j + 1          (33)
    Fin
Fin     i = i + 1          (34)
```

Figure 1

7. Résultats expérimentaux:

On programmé en langage C parallèle les versions séquentielles et parallèles respectives des algorithmes. En demandant la valeur de l'horloge on a pu mesurer:

t seq = le temps d'exécution en mode séquentiel sur un processeur.

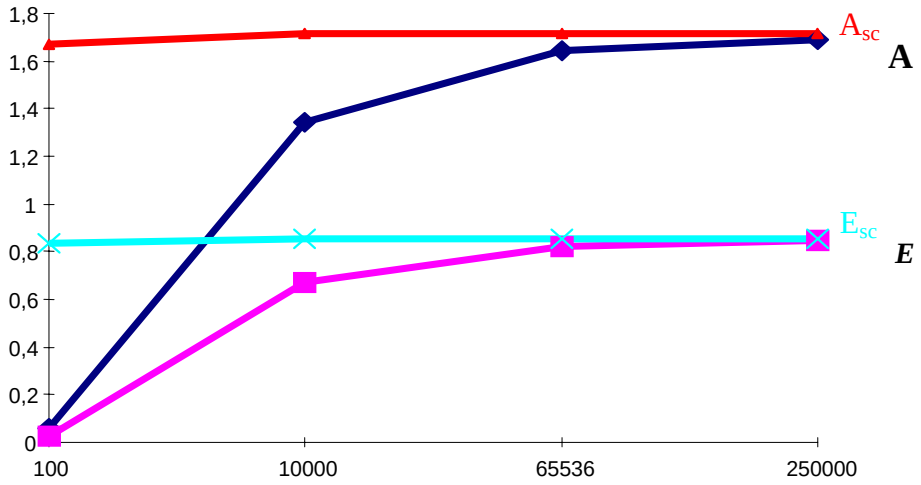
$t_{//}$ = le temps de calcul en parallèle + le temps de communication.

$t_{// sc}$ = le temps d'exécution parallèle sans communication où $t_{// sc} = \max (t_{// sc})_{i=1,8}$

Les tableaux ci-dessous présentent les résultats pratiques obtenus pour les deux algorithmes implémentés.

a) Cas sur deux processeurs (algorithme de DERICHE):

Taille de la matrice	t seq en secondes	$t_{//}$ en secondes	$t_{//sc}$ en secondes	A	E	A_{sc}	E_{sc}
100	0.062	1.061	0.037	0.0584	0.0292	1.67	0.835
10.000	6.342	4.722	3.699	1.343	0.671	1.714	0.857
65536	41.568	25.266	24.243	1.645	0.822	1.714	0.857
250.000	158.573	93.778	92.479	1.690	0.845	1.714	0.857



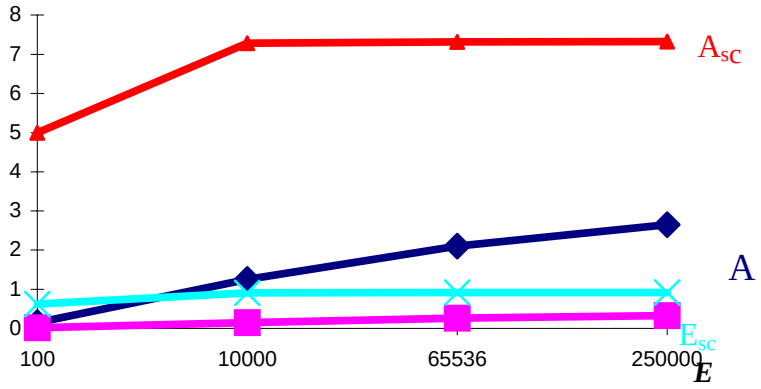
Les résultats précédents montrent que si la taille de la matrice A augmente, le temps de communication sera couvert par le temps de calcul.

A_{sc} et E_{sc} représentent respectivement l'accélération et l'efficacité sans surcoût du temps de communication.

Nous considérons alors que la solution choisie apporte un bénéfice souhaitable et la performance se trouve valable. En effet, E ≈ 1 (cas idéal = 0) et A est proche du nombre de processeurs (2).

b) Cas sur 8 processeurs (algorithme utilisant les opérateurs locaux de dérivation):

Taille de la matrice	t seq en secondes	$t_{//}$ en secondes	$t_{//sc}$ en secondes	A	E	A_{sc}	E_{sc}
10×10	0.01	0.069	0.002	0.145	0.018	5	0.62
100×100	1.26	1.008	0.173	1.25	0.15	7.28	0.91
256×256	8.363	3.990	1.143	2.096	0.262	7.316	0.914
500×500	32.025	12.106	4.374	2.645	0.33	7.321	0.915



D'après les résultats obtenus le surcoût de communication est assez important; par conséquent, le bénéfice souhaitable ne peut être considéré comme acquis que si l'on exclue la pollution due aux échanges de données, de l'accélération et de l'efficacité.

8. Conclusion

Les caractéristiques des algorithmes considérés sont déterminantes dans la décomposition en tâches de l'algorithme séquentiel et dans la méthode de parallélisation. Toutefois, cette méthode peut être appliquée à tout algorithme présentant les mêmes caractéristiques .
D'autre part, le problème de la décomposition en tâches d'un algorithme n'est pas simple car il n'existe pas de méthode systématique et pour l'instant, la solution dépend de l'habilité du concepteur. En effet, la difficulté tient essentiellement à

déterminer les critères de décomposition. La démarche suivie dans notre travail nous a permis de réaliser une décomposition fonctionnelle d'un algorithme, décrite en terme de tâches et de relations entre tâches.
La modélisation de la décomposition obtenue par un graphe de dépendance a visualisé un ordre partiel d'exécution, induit par une relation de précédence dans le temps.
Nous avons défini alors, entre les tâches, des relations de dépendance et une distance de dépendance, ce qui nous a permis d'obtenir l'exécution simultanée de certaines tâches.
Enfin, il était nécessaire de situer les gains de performances par la mesure de l'efficacité du programme parallélisé en réalisant le projection de ce dernier sur une machine parallèle (TN310). En effet, cette projection a consisté à faire exécuter les tâches dans un ordre compatible avec le graphe de communication des tâches obtenu.

Bibliographie

[A 93] ANDEAS KÖPPEL : " Parallélisation des Algorithmes de Traitement d'Images sur Réseau de Transputers "; Rapport de la théorie; ESIEE; janvier 1993.

[AFRVRRV 94] G.AUTHIE, A.FERREIRA, J.L.ROCH, G.VILLARD, J.ROMAN, C.ROUCAIROL, B.VIROT : " Algorithmes Parallèles, Analyse et Conception "; Editions HERMES; Paris 1994.

[B 95] FENGO BO : " Algorithmes de Placement de Tâches et leurs Parallélisations dans les Architectures Multiprocesseurs sans Mémoire Partagée "; Thèse de doctorat; Université Paris VI; 1995.

[BGS 94] DAVID F. BACON, SUSAN L.GRANAM, OLIVIER J. SHARP: "Compiler Transformations for High-Performance Computing "; University of California.Berkeley; Decembre 1994.

[CMAR 92] J.M.CHASSERY, G.MANDOU, D.ADELH, J.L.ROCH : "Implantation des Algorithmes Bas Niveau de Traitement d'Images sur un Réseau de Transputers", La lettre du Transputer N° 13; p 55-69; mars 1992.

[DD 92] DAPOIGNY.R, A.DIOU : " Parallélisation d'un Algorithme de Filtrage Récurrent sur Différentes Architectures Parallèles. Etude Comparative des Performances "; La lettre du Transputer N° 13; p.35-52; mars 1992

[DS 95] ROYEN DAMIEN, SCHROELER DAVID : " Parallélisation de Traitement d'Images sur Réseau de Transputers "; Projet ESIEE I4; 1995.

[Ghoul 95] S. GHOUL, M.S. BENDELLOUL, D. MESLATI, Z.E. BOURAS: " Un Modèle Multi-Agents de Dépendances de Programme "; Rapport interne N°5; Projet de Recherche MERCI; Université de Annaba; 1995

[HEN 89]: A. HENNI " Conception et réalisation d'un système de communication en environnement multiprocesseurs "; Thèse de doctorat; Université Paris VI; 1989.

[HL] H.HOGEN, B.PORCHER LABREUILLE: " Etude sur la Parallélisation d'Algorithmes de Traitement d'Images " ; Tome 2 ; Contrat ETCA, N° 8205/346.

[JR 94] DANIEL JACKSON, EUGENE J.ROLLINS: " A New Model of Program Dependances for Reverse Engineering "; Volume 13 n° 5, Carnegie Mellon University; Decembre 1994.

[K 73] R.M.KELLER : " Parallel Program Shematic and Maximal Parallelism "; J.ACM 203; pp514-537; July 1973.

[M 94] SALOTTE JEAN MARC : " Gestion des Informations dans les Premières Etapes de la Vision par Ordinateur "; Thèse de doctorat; Inst. Nat. Polytech. de Grenoble; 1994.

[P 94] DWAYNE PHILLIPS : " Analysing and Enchancing Digital Images "; R & D Publications; Inc. Lawrance, Kansas; 1994.

[PW 92] VICTOR K.PRASANNA, CHO-LI WANG : " Parallélism for Image Understanding " University of Minnesota A.H.P.C.R.Centre; pp 1042-1068; 1992.

[S 92] JEAN-PAUL SANSONNET : " Architectures des Machines Parallèles "; chapitres 1,2,4,5,7 ; LRI-Archi, Université de Paris XI; 1990-1992.

[SAOULI 98] R. SAOULI : "Méthodes de Parallélistion des Algorithmes de Traitement d'Images " Mémoire de Magister en Informatique , Université de Batna 1998

[Z 92] NIZAR ZARKA: " Conception d'un Circuit Intégré de Détection Optimale de Contours "; Thèse de doctorat; Université Paris 6; pp 9-41; 1992.

.