République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de La Recherche Scientifique
Université Mouloud MAMMERI de Tizi-Ouzou
Faculté du Génie électrique et Informatique
Département Informatique

Mémoire de Fin d'Études
de MASTER ACADEMIQUE
Domaine : **Mathématiques et Informatique**
Filière : **Informatique**
Spécialité : **Systèmes Informatiques**

Thème : Traitement parallèle pour l'extraction de contours d'image dans un environnement multi cœurs

Proposé et dirigé par :
Mr L.DJEMA

Réalise par :
Mr Massinissa HASSANI
Mr Ali BOUZEBOUDJA

2015/2016

To ...

Table des matières

Liste des figures

Liste des Tables

Introduction générale

Depuis les débuts de l'informatique la puissance de calcul des processeurs n'a cessé de croitre. Au début cela se faisait en augmentant la fréquence d'horloge. Pour les développeurs, l'augmentation de la fréquence d'horloge des processeurs est synonyme d'amélioration des performances de leurs programmes sans en modifier la moindre ligne de code. Mais cet augmentation de fréquence d'horloge est devenue quasiment impossible à cause des contraintes matérielles telles que la consommation d'énergie, ainsi que la chaleur dégagée, qui est devenue très importante et très difficile à dissiper.

Les concepteurs de processeurs ont adopté une autre approche, au lieu d'essayer d'augmenter la fréquence d'horloge, l'idée est de multiplier le nombre de cœurs au sein d'un seul processeur. Mais les applications séquentielles existantes ne pourront plus bénéficier automatiquement des performances de ces processeurs multi cœurs sans exploiter tous les cœurs disponibles sur le processeur. Ce qui signifie que les développeurs doivent se penche sur la programmation parallèle, qui représente une nouvelle approche dans le développement des logiciels. Cependant, la conception d'un programme parallèle est plus complexe que la conception d'un programme séquentiel.

Le traitement d'images et la vision par ordinateur sont des domaines qui requièrent une grande puissance de calcul et qui se prêtent parfaitement au traitement parallèle. L'extraction de contours d'image est un champ qui appartient à la fois au traitement d'image et à la vision par ordinateur. Le but de cette technique est de réduire la quantité de données et éliminer les informations qu'on peut juger moins pertinentes, tout en préservant les propriétés structurelles importantes de l'image.

L'objectif de notre travail est d'exploiter le parallélisme dans un environnement

multi cœurs pour l'extraction des contours d'image. Notre mémoire est organisé en quatre chapitres. Dans le premier chapitre, nous donnons un aperçu sur le parallélisme et les processeurs multi cœurs. Nous consacrons le deuxième chapitre à la présentation de l'outil Intel TBB de Intel, conçu pour facilité le développement d'applications parallèles. Dans le chapitre III, nous exposons des généralités sur le traitement d'image et la méthode de Dériche pour l'extraction de contours d'image. Enfin, dans le chapitre IV nous présentons notre réalisation sur l'application d'extraction de contours d'image et l'expérimentation réalisée sur des images médicales de différentes capacités.

Chapitre I

Parallèlisme et Architecture multi cœurs

I.1 Introduction

Traditionnellement, les logiciels ont été programmé pour une exécution séquentiel, un problème est divisé en une série d'instructions qui sont exécutées séquentiellement l'une après l'autre sur un seul processeur qui ne peut exécuter qu'une seule instruction à la fois.
Mais le besoin croissant de rapidité de calcul dans les applications scientifiques modernes a conduit a l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème. Ces ressources de calcul peuvent être une seule machine avec plusieurs processeurs, un nombre quelconque de machines connectées via un réseau ou alors une combinaison des deux. Ce qui permet l'utilisation des ressources non locaux lorsque les ressources locales sont insuffisantes

Le problème est décomposée en plusieurs parties qui peuvent être résolues simultanément et chaque partie est décomposée en une série d'instructions, ainsi les instructions de chaque partie peuvent être exécutées indépendamment sur différent processeurs. Ce qui permet aussi de résoudre des problèmes de plus en plus complexes.

Les avancées technologiques des dernières années, notamment dans les architectures des processeurs, et la disponibilité des machines parallèles au grand public avec des processeurs multi cœurs dans la plupart des ordinateurs personnels, montrent

clairement que le parallélisme est l'avenir de l'informatique.

I.2 Concepts généraux sur les architectures des ordinateurs

I.2.1 Processeur mono cœur

a) Architecture de Von Neumann

C'est le modèle qui fut inventé par le mathématicien hongrois John von Neumann qui a posé les premières bases de la conception d'un ordinateur en 1945. A partir de ce moment, la majorité des ordinateurs ont été conçus sur ces bases.
L'architecture von Neumann (figure I.1) est constituée de 4 composants principaux :

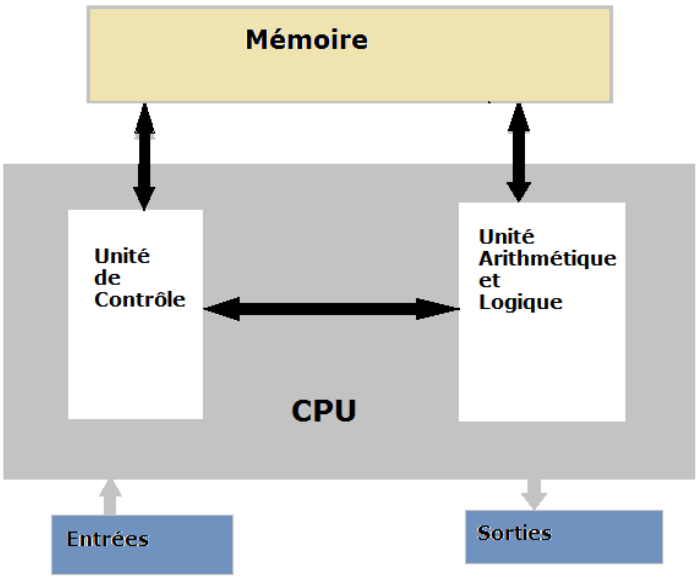


FIGURE I.1 – L'architecture Von Neumann

Une mémoire : La mémoire à accès aléatoire (RAM) en lecture/écriture est utilisée pour stocker les instructions ainsi que les données.

Une unité de contrôle : L'unité de contrôle va chercher les instructions ou les données de la mémoire, décode les instructions et coordonne séquentiellement les opérations afin d'accomplir la tâche programmée.

Une unité arithmétique et logique (ALU) : L'ALU effectue les opérations arithmétiques de base.

Des entrées/sorties (I/O) : Les I/O qui constituent une interface pour l'échange de données avec les différents périphériques.

I.2.2 Augmentation les performances

Les concepteurs de processeurs ont pu améliorer les performances des architectures dans trois domaines :

a) Optimisation de l'exécution :

Cela se traduit par l'accomplissement de plus de travail en un cycle d'horloge. Parmi les technique les plus connues : le piplining.

b) La fréquence d'horloge :

Cela consiste à augmenter la capacité du processeur a supporter une fréquence d'horloge plus élevé.

c) La mémoire cache :

Ajouter la mémoire cache sur un processeur a pour but de réduire le temps d'accès aux données. En effet, les accès à la mémoire centrale se font toujours avec des latences très grandes, et réduire le temps d'accès aux données demeure une excellente technique pour améliorer les performances. Les tailles des caches ont augmenté sensiblement durant les dernières années pour atteindre des valeurs autour de 10 Mo sur les dernières générations de processeurs[1].

I.2.3 La mémoire cache

La mémoire cache est la mémoire la plus rapide dans la hiérarchie des mémoires et se rapproche de la vitesse des composants CPU. Tout processeur actuel contient des mémoires caches, qui rendent les accès à la mémoire plus rapides. Sur les processeurs multi cœurs, l'organisation de ces mémoires caches est adaptée à la présence de plusieurs cœurs.

a) Fonctionnement

Quand un cœur a besoin d'une donnée, alors le cache est examiné en premier. Si la donnée s'y trouve, la donnée est lue rapidement (succès de cache), sinon (défaut de cache) la donnée sera lue à partir de la mémoire centrale.

b) Niveau de cache

On différencie plusieurs niveaux de caches, souvent au nombre de trois :

Le cache de premier niveau (Ll) : La plus rapide et de plus petite capacité, souvent cache de données est séparé du cache d'instructions .

Le cache de second niveau (L2) : moins rapide, plus grande capacité que le cache L1.

Le cache de troisième niveau (L3) : encore moins rapide, et encore plus grande capacité que le cache L2, et peut être situés dedans ou hors du processeur.

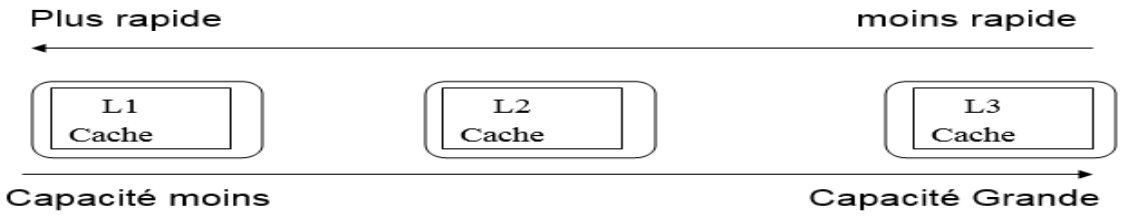


FIGURE I.2 – Niveaux des caches

I.2.4 le pipelining

Pendant l'exécution d'un programme chaque instruction est exécutée en plusieurs étapes (chercher, décoder, exécuter ...) et l'ensemble de ces étapes peuvent prendre plusieurs cycles d'horloge, or l'unité de calcul n'est utilisée que pendant l'étape d'exécution.
Pour augmenter le débit des instructions exécutées, les fabricants ont mis en place le technique de pipelining. L'idée est de pouvoir exécuter plusieurs étapes indépendantes simultanément, par exemple exécuter une instruction tout en décodant l'instruction suivante.
Cela nécessite d'organiser le processeur en étages, chacun capables d'exécuter une micro-opérations d'une façon indépendante.

I.3 Classification de Flynn des machines parallèles

Il existe plusieurs manières de classer les machines parallèles. Toutefois, il existe une classification qui est largement utilisée depuis 1966 et qui est celle de Flynn. Cette classification distingue les architectures parallèles selon deux paramètres indépendants qui sont le flux d'instruction et le flux de donnée : chacun de ces deux paramètres peut avoir deux états possibles Single ou Multiple. Le tableau I.1 illustre la classification de Flynn[1].

	Single data	Multiple data
Single instruction	SISD	SIMD
Multiple instruction	MISD	MIMD

TABLE I.1 – Classification de Flynn des machines parallèles

I.3.1 Single instruction, Single data (SISD)

C'est le type de machines le plus ancien.Une machine séquentielle qui ne peut exécuter qu'un seul flux d'instructions en un cycle d'horloge. De plus, un seul flux de données est utilisé comme entrée en un cycle d'horloge.

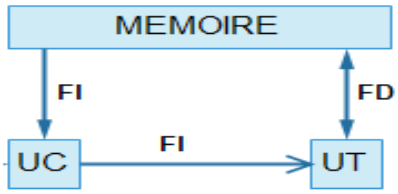


FIGURE I.3 – Single instruction, Single data (SISD)

FI=Flux d'instruction. FD=Flux de données. UC=Unité de contrôle. UT=Unité de Traitement.
Dans ce type de machine on ne peut réaliser qu'un parallélisme au niveau instructions (le Piplining).

I.3.2 Single instruction, multiple data (SIMD)

Machine parallèle dont les processeurs exécutent la même instruction en un cycle d'horloge donné. Cependant, chaque unité de traitement peut opérer sur un élément de données différent. Ce type de machines est bien adapté pour des problèmes régu-

liers tels que le traitement d'images et le rendu graphique. La plupart des processeurs graphique adoptent cette architecture.

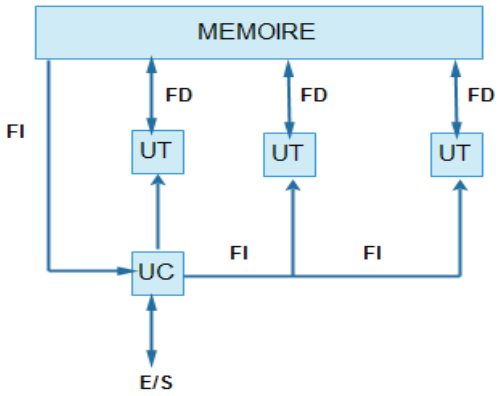


FIGURE I.4 – Single instruction, multiple data (SIMD)

I.3.3 multiple instruction, single data (MISD)

Un seul flux de données alimente plusieurs unités de traitement et chaque unité de traitement opère sur les données un traitement différent grâce à un flux d'instruction indépendant. Peu d'exemples concrets de cette architecture a été réalisé.

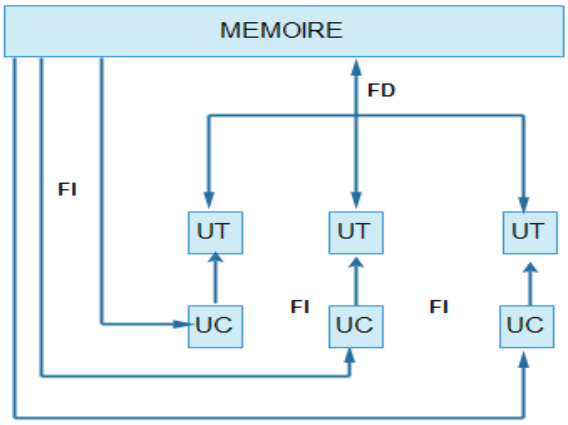


FIGURE I.5 – multiple instruction, single data (MISD)

I.3.4 multiple instruction, multiple data (MIMD)

Actuellement c'est le type le plus répandu des machines parallèles. Dans ces machines chaque processeur peut exécuter un flux d'instructions différent et peut traiter un flux de données différent. Actuellement la plupart des ordinateurs et superordinateurs [1] entrent dans cette catégorie. On peut citer les grilles de calculs [2], les multi-processeurs SMP(Symetric Multi-Processor) et les processeurs multi cœurs.

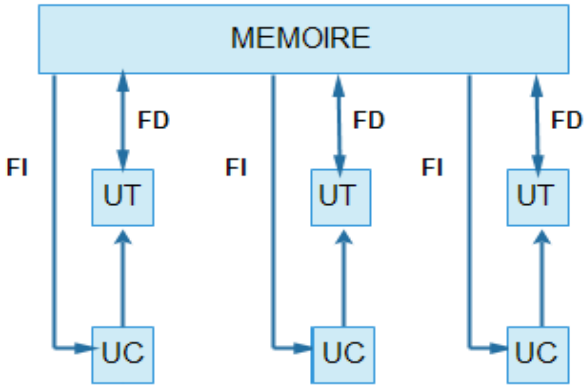


FIGURE I.6 – multiple instruction, multiple data (MIMD)

I.4 Architectures mémoire des machines parallèles

I.4.1 Les machines parallèles à mémoire partagée

Toutes les machines de cette catégorie partagent une propriété commune qui est la possibilité à tous les processeurs d'accéder à la mémoire comme un espace d'adressage global. Ainsi, plusieurs processeurs peuvent opérer d'une manière indépendante mais partagent la même ressource mémoire. Un changement opéré par un processeur dans un emplacement mémoire est visible à tous les autres processeurs. Cette classe de machines peut être divisée en deux sous-classes basées sur les temps d'accès à la mémoire : UMA et NUMA.

Uniform memory access (UMA) : Ce sont principalement les machines de type SMP qui possèdent plusieurs processeurs identiques et qui peuvent accéder de la

1. un ordinateur conçu pour atteindre les plus hautes performances possibles.
2. une infrastructure virtuelle constituée d'un ensemble de ressources informatiques potentiellement partagées, distribuées, hétérogènes, délocalisées et autonomes.

même manière et en un temps identique à la mémoire. Elles sont parfois appelées CC-UMA - Cache Cohérent UMA. La cohérence de cache signifie que si un processeur met à jour un emplacement de la mémoire tous les autres processeurs sont au courant de ce changement. Cette fonctionnalité est assurée au niveau matériel[2] .

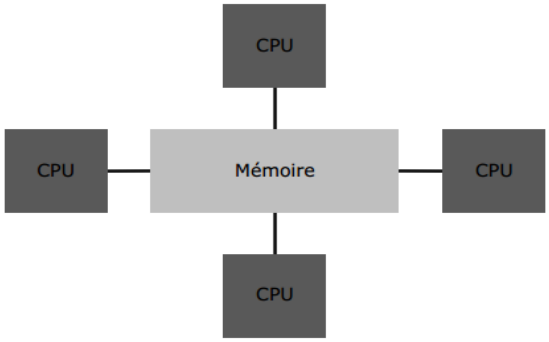


FIGURE I.7 – Mémoire partagé (UMA)

Non-uniform memory access (NUMA) : Ce type de machines est souvent conçu en connectant deux ou plusieurs SMPs,et chaque SMP peut avoir un accès direct à la mémoire d'un autre SMP. Le temps d'accès à une mémoire donnée n'est pas le même pour tous les processeurs et lorsque l'accès est fait a travers un lien de connexion , l'accès est plus lent. Si la cohérence de cache est garantie on parle également de CC-NUMA.

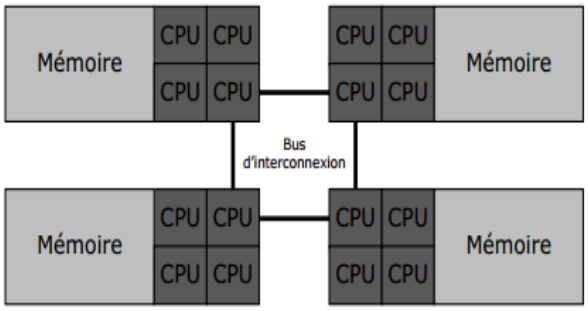


FIGURE I.8 – Mémoire partagé (NUMA)

a) Avantages

- L'espace d'adresse global facilite la programmation.

- Le partage des données entre les tâches est rapide et uniforme à cause de la proximité de la mémoire au CPUs.

b) Inconvénients

- Manque de scalabilité : augmenter le nombre de CPUs accroît le trafic sur le chemin d'accès à la mémoire partagée.
- Le programmeur doit gérer la synchronisation pour un accès correct à la mémoire globale.
- La conception et la production des machines à mémoire partagée sont difficiles et coûteux avec un grand nombre de processeurs.

c) Modèles de programmation associés

Le modèle à mémoire partagée : Les implémentations de ce modèle sur les machines à mémoire partagée se résument au compilateur natif qui traduit les variables du programme en adresses mémoire globales.

Le modèle de programmation par threads : Les implémentations des threads comportent généralement :

- Une bibliothèque de fonctions.
- Une série de directives de compilation.

Dans les deux cas le programmeur est responsable de la définition du parallélisme. Il existe plusieurs implémentations des threads :

Intel TBB : Intel TBB(Threading Building Blocks) est une bibliothèque publiée par Intel en 2007. Développée en C++, elle permet d'abstraire au maximum les détails délicats de la programmation parallèle´en utilisant les concepts objets(classes, template). Fonctionne sur différents compilateurs(Intel, Microsoft, gcc), adapté aux architectures multicœur en particulier .

JAVA ParaTask : ParaTask est un compilateur java parallèle, qui utilise les concepts du parallélisme dans un environnement orienté objet, le programmeur introduit le parallélisme avec des simples mots clé. Il supporte une approche intuitive de gestion des dépendances en se basant sur les applications GUI[3].

OpenMP : est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Supportée sur de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement[4].

Thread POSIX : Souvent appelés pthreads, un sous-standard de la norme POSIX décrivant une interface de programmation permettant de gérer des threads. Elle définit un ensemble de types de données, de fonctions et de constantes en langage C[4].

I.4.2 Les Machines parallèles à mémoire distribuée

Les machines à mémoire distribuée requièrent un réseau de communication pour connecter des processeurs possédant leur propre mémoire locale. Les adresses mémoire d'un processeur donné ne correspondent pas à celles d'un autre et par conséquent le concept de mémoire globale n'existe pas. Puisque chaque processeur possède sa propre mémoire privée. Chaque changement opéré sur sa mémoire n'a aucun effet sur la mémoire des autres processeurs ce qui exclue le concept de cohérence de cache. Lorsqu'un processeur à besoin des données contenues dans la mémoire d'un autre processeur, le programmeur est en charge de définir quand et comment les données sont transférées. Ce dernier est aussi responsable de la synchronisation.

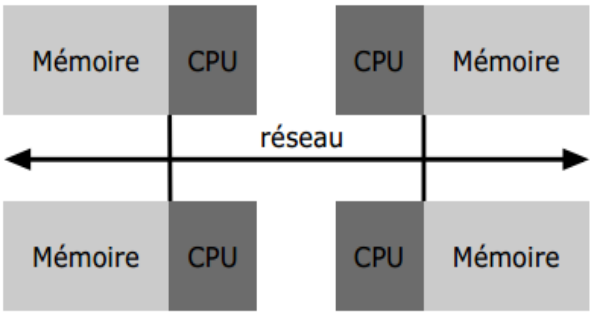


FIGURE I.9 – Mémoire distribuée

a) Avantages

- L'augmentation du nombre de processeur augmente la mémoire.

- Accès rapide à la mémoire locale sur chaque processeur, sans surcoût de gestion de cohérence de cache.

b) Inconvénients

- Le programmeur doit gérer toutes les communications entre processeurs.
- Temps d'accès mémoire non locaux élevés.

c) Modèles de programmation associés

Le modèle passage de message : De nos jours, **MPI**[3] et **PVM**[4] sont les implémentations les plus utilisées pour le passage de message.

MPI : Conçue en 1993-94, est une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages.

PVM : Développé à Oak Ridge National Laboratory en 1989, a servi de base à un projet plus vaste commun à trois universités des Etats-Unis, est un ensemble de bibliothèques en langages C et Fortran qui permet d'agréger un réseau d'ordinateurs en un seul ordinateur virtuel,d'où l'appellation de machine virtuelle parallèle[4].

I.4.3 Les Machines parallèles à mémoire hybride(Distribuée-partagée)

Ce type de machines regroupent les deux types précédents : partagée et distribuée. La composante partagée est une machine à mémoire partagés et souvent une machine SMP. La composante distribuée est la mise en réseau de plusieurs machines à mémoire partagée qui ne peuvent adresser que leur propre mémoire et le transfert de données entre ces dernières requiert des communications à travers le réseau.
La différence majeure entre ce type de machines et les machines à mémoire partagée NUMA , est le fait que l'espace mémoire n'est pas partagé et le programmeur est en charge de définir quand et comment les données sont transférées entre les machines a mémoire partagée.
Les machines les plus importantes et les plus rapides dans le monde aujourd'hui utilisent cet architecture.[2]

3. Message Passing Interface
4. Parallel Virtual Machine

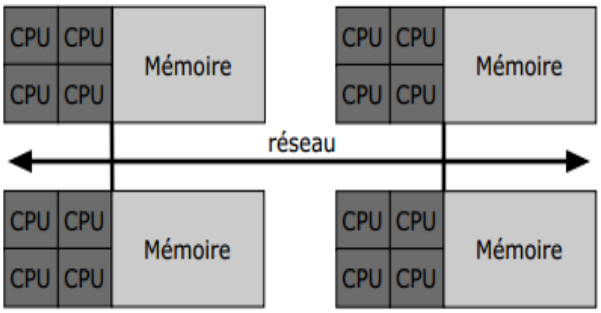


FIGURE I.10 – Mémoire hybride

a) Modèles de programmation associés

Dans le cas de cet architecture mémoire une combinaison des modèles de programmation à mémoire partagée et à mémoire distribuée est utilisée.

I.5 Le parallélisme

I.5.1 Les types de parallélisme

a) Parallélisme de contrôle

Dans ce type de parallélisme des taches indépendants entre elles sont exécutées simultanément sur plusieurs processeurs.Donc le programme doit contenir des séquences d'opérations indépendantes qui peuvent être exécutées en parallèle.

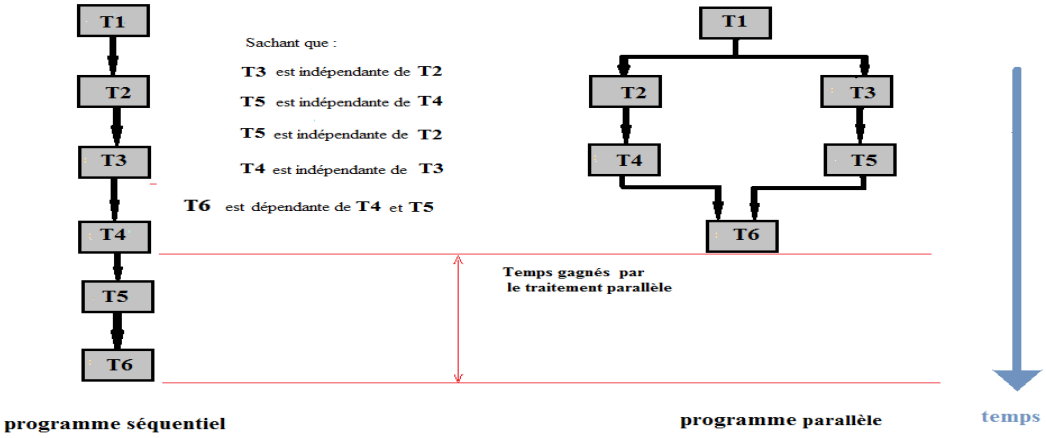


FIGURE I.11 – Parallélisme de contrôle

b) Parallélisme de données

Si un programme utilise un grand nombre de données similaires (Vecteurs, Matrice) sur lesquelles il effectue le même type d'équations, on dit qu'il utilise des données régulières. Cette forme de parallélisme consiste a décomposer une structure de données régulière, et repartir les differentes parties de la structure su les différents processeurs, ensuite la même opération est effectuée par chaque processeur sur ces partie.
Il existe trois type de décomposition de données :

Décomposition par blocs : Le nombre de données N est devisé par le nombre de processeur P, ce qui donne N/P données par chaque processeur.

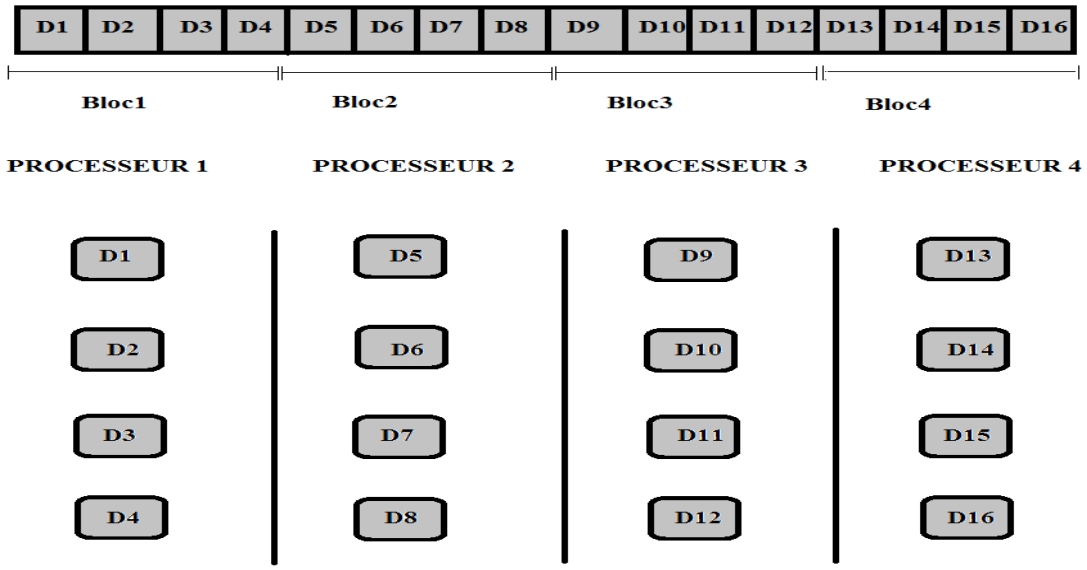


FIGURE I.12 – Décomposition par blocs

Décomposition cyclique : Les données sont distribués d'une manière cyclique sur les processeurs(donnée par donnée).

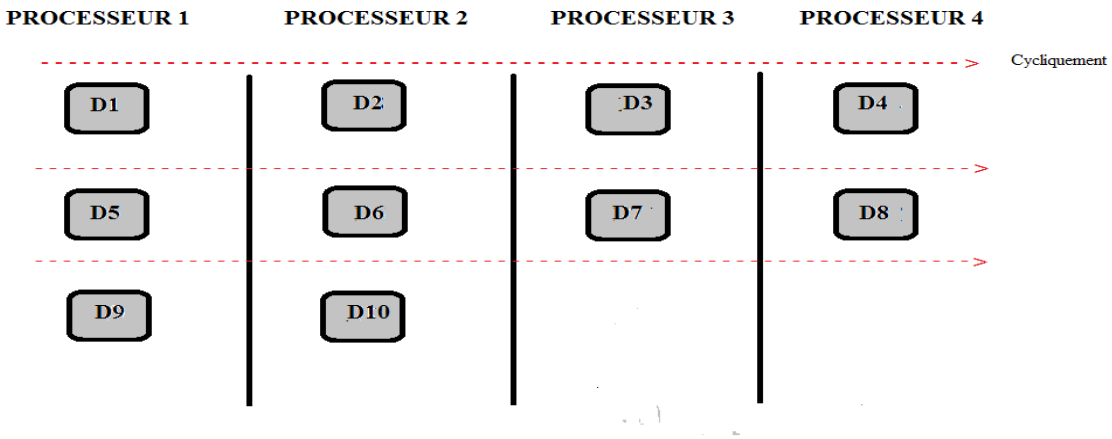


FIGURE I.13 – Décomposition cyclique

Décomposition par blocs cyclique : La décomposition se fait en précisant la taille des blocs a distribuer , puis on distribue les blocs d'une manière cyclique sur les processeurs.

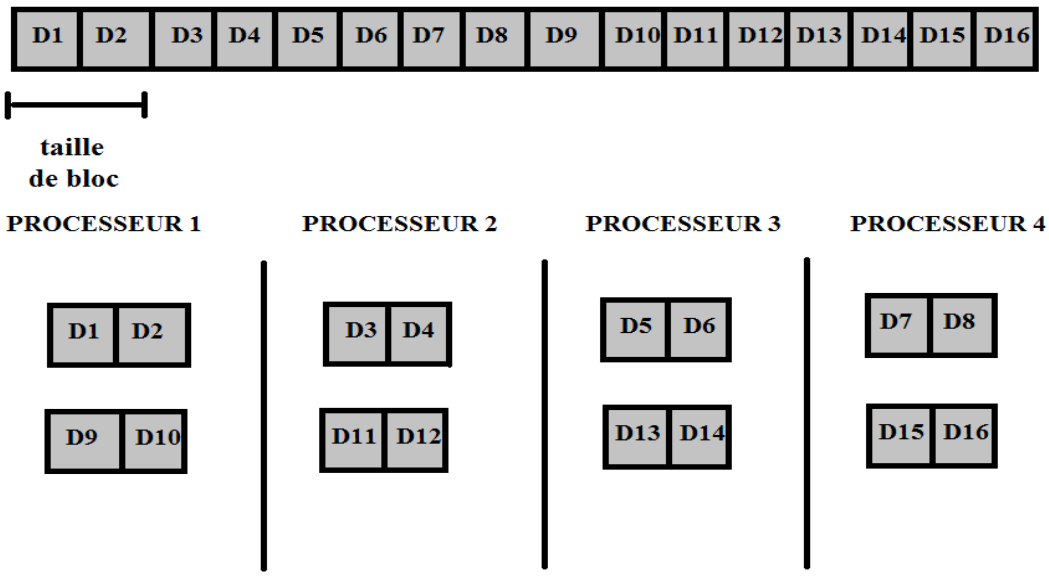


FIGURE I.14 – Décomposition par blocs cyclique

c) Parallélisme de flux

Ce parallélisme reprend le concept du pipeline. Le flux de données est traité en une suite d'opération en cascade. Les résultats d'opération de l'étape N sont transmis à une autre unité qui mène en charge les opérations de l'étape N+1. Le gain obtenu par ce modèle de programmation parallèle est égale au nombre d'étages du pipeline.

Temps	Tache 1	Tache 2	Tache 3	Tache 4	Tache 5
t1	D 1				
t2	D 2	D 1			
t3	D 3	D 2	D 1		
t4	D 4	D 3	D 2	D 1	
t5	D 5	D 4	D 3	D 2	D 1
t6	D 6	D 5	D 4	D 3	D 2

TABLE I.2 – Parallélisme de flux

I.5.2 La Granularité du parallélisme

La granularité du parallélisme est définie comme étant le ratio calcul/communication. Il existe deux formes de granularité :

a) Parallélisme à grain fin :

Dans ce cas le ratio calcul/communication est faible, les tâches passent un temps important en communications et pas en calcul, ce qui facilite l'équilibrage des charge, mais réduit les performances.

b) Parallélisme à gros grain :

Dans ce cas le ratio calcul/communication est important, les tâches passent la majorité du temps en calcul et pas en communications ce qui augmente les performances, mais rend l'équilibrage des charge difficile .

I.5.3 Dépendances de données

a) Dépendances de flux

Les dépendances de données sont un des principaux inhibiteurs de la parallélisation. Une dépendance de données existe lorsqu'une modification de l'ordre d'exécution des instructions change le résultat du programme.

Dépendance avant :

```
for (i=0, i<n , i++){
        a(i)=a(i+1) ;
  }
```

Dépendance arrière :

```
for (i=0, i<n , i++){
        a(i)=a(i-1) ;
  }
```

Réduction :

```
s=0 ;
for (i=0, i<n , i++){
```

```
        s=s+a(i) ;
    }
```

b) Dépendances de contrôle

```
For (i = 0, i < n, i + +){
    if (a(i) > 0) goto etiquette ;
     a(i)=a(i) + k ;
 }
etiquette
```

Cette boucle contient un branchement de sortie, le flux de contrôle peut donc sortir de la boucle soudainement.il n'existe pas de moyens simples de paralléliser automatiquement cette boucle.

I.5.4 Conception d'un programme parallèle

La conception d'un programme parallèle pour un problème donné, est plus complexe que la conception d'un programme en séquentiel du même problème, car il y a des facteurs en plus qu'il faut prendre en considération.

a) Comprendre le problème

La première question qui se pose avant de commencer à développer un programme parallèle pour un problème donné est celle de la faisabilité d'une telle solution. En effet, il existe certains problèmes dans lesquels il n'existe aucune forme de parallélisme exploitable. Pour rependre a cette question il faut Comprendre le problème. Si le problème s'agit de la parallélisation d'un programme séquentiel, il faut comprendre son code.

b) Partitionner le problème

Le problème doit être partitionné selon la nature du parallélisme contenu dans le problème : parallélisme de données ou de tâches.

c) Gestion des communications

Le fait de décomposer le problème engendre parfois un besoin de communication entre les tâches, si des communications sont nécessaires, il est alors souvent utile de

regrouper les transferts en un seul bloc, ce qui n'est pas tout le temps possible, car la multiplicité des transferts engendre autant de temps de latence que de transfert.

d) Gestion de la synchronisation

Le fait de décomposer le problème engendre aussi un besoin de synchronisation entre les tâches. Les tâches ont souvent besoin de se synchroniser soit pendant un échange de données, soit à la suite d'une opération collective. Parmi les opérations de synchronisation les plus utilisées on trouve les barrières, les sémaphores et les verrous.

e) Équilibrage de charge

Une distribution équitable des charges de travail sur les processeurs est nécessaire afin de minimiser les durées d'inactivité des processeurs. Lorsque les tâches effectuent le même travail, l'équilibrage de charge est trivial, il suffit d'attribuer aux tâches les mêmes quantités de données. Si par contre les tâches exécutent un code différent, un ajustement de la charge de travail est nécessaire.

I.5.5 Évaluation des performances

Pour l'évaluation des performances des programmes parallèles, deux grandeurs ont été définie : l'accélération et l'efficacité.

a) Acceleration :

Chaque programme parallèle est caractérisé pas son facteur d'accélération Acc qui une grandeur sans dimension désignée par le rapport entre le temps du l'algorithme séquentiel et le temps de l'algorithme parallèle résolvant le même problème sur une machine parallèle donnée.

$$Acc = \frac{T_s}{T_p} \qquad 0 \leq Acc \leq N$$

Ou T_s est le temps d'exécution séquentiel du programme.
Et T_p est le temps d'exécution parallèle du programme sur N processeurs.

Loi d'Amdahl : La loi d'Amdahl stipule que l'accélération d'un programme est limitée par la proportion parallélisable de celui-ci. Cela se traduit par l'équation

suivante[1] :

$$Acc = \frac{1}{(1 - FP) + \frac{FP}{N}}$$

Accélération pour un nombre maximale de processeurs :

$$\lim_{N \to \infty} Acc = \lim_{N \to \infty} \frac{1}{(1 - FP) + \frac{FP}{N}} = \frac{1}{1 - FP}$$

Où FP est la fraction parallélisable du programme
Et N le nombre de processeurs.
On peut constater à partir de l'équation que l'accélération est limitée par la proportion de code parallélisable. Cela signifie qu'une augmentation linéaire de l'accélération n'est plus possible à partir d'une certaine valeur de N et qu'il est inutile d'ajouter des unités de traitement pour améliorer les performances.

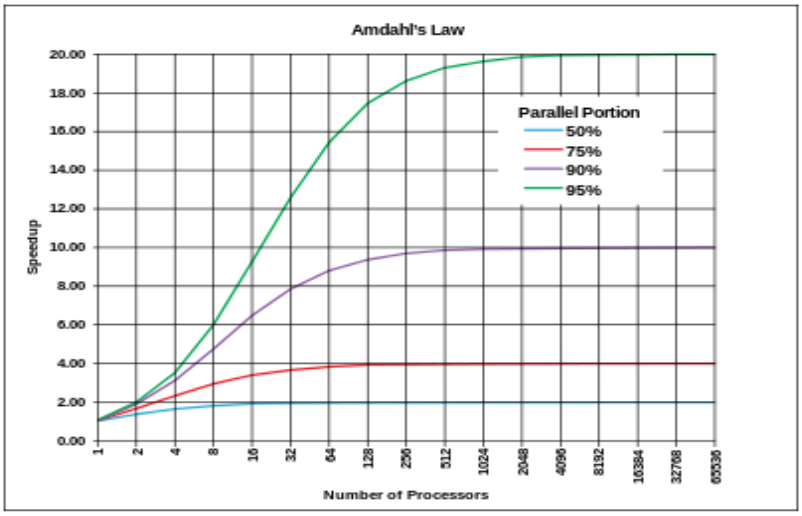


FIGURE I.15 – Évolution de l'accélération par rapport au nombre de processeur et la proportion parallélisable d'un programme.

b) Efficacité :

L'efficacité Eff traduit le taux d'utilisation des N processeurs de la machine parallèle désignée par le rapport entre l'accélération Acc et le nombre de processeurs.

$$Eff = \frac{T_s}{T_p * N}$$

c) Scalabilité ou passage à l'échelle :

Capacité d'un algorithme parallèle à passer à l'échelle sur une architecture donnée. On aimerait voir évoluer les performances de l'application de façon linéaire avec le nombre de processeurs. Il est difficile de définir la scalabilité d'un algorithme parallèle sans tenir compte de l'architecture matérielle utilisée et inversement de parler de scalabilité d'une architecture parallèle indépendamment des algorithmes.[5]

I.6 Architecture multi cœurs

I.6.1 Motivations

Pour augmenter la puissance des processeurs les constructeurs augmentait la fréquence de traitement de leurs processeurs. Mais cette méthode a fini par atteindre ses limites. En effet, l'augmentation de fréquence cause rapidement des problèmes de surchauffe. D'autre part le besoin croissant en énergie des processeurs était problématique notamment pour les ordinateurs portables car la puissance dissipée serait quadruplée si on doublait la fréquence d'horloge, alors qu'elle serait doublée si on double le nombre cœurs.
Pour pallier à ces difficultés et atteindre les performances dont on a besoin, au début la solution était de construire des machines multiprocesseurs [5], et dans la dernière décennie la préférence va vers l'utilisation des processeurs multi cœurs.

I.6.2 Processeur multi cœurs

Un processeur multi cœurs est un processeur possédant plusieurs cœurs physiques fonctionnant simultanément. Toutes les fonctionnalités nécessaires à l'exécution d'un programme sont présentes dans ces cœurs : compteur ordinal, registres, unités de calcul, etc. Des caches sont définis pour chaque cœur dédiés ou partagés entre eux[4].

Le premier processeur multi cœurs a été commercialisé par IBM (POWER4 en 2001). Les premiers exemplaires de processeurs multicœurs d'Intel et d'AMD sont arrivés sur le marché des ordinateurs personnels en 2005.

5. Une machine multi-processeurs est une machine qui est équipés de plusieurs processeurs. Les processeurs sont connectés au moyen d'un bus à une mémoire principale partagée. Dans le cas ou tous les processeurs de la machine sont identiques, on parle des **SMP** (Symmetric Multi Processor).

I.6.3 Multi-Threading

Le changement du contexte des processus en exécution est très coûteux en matière du temps, car il faut sauvegarder le contexte d'exécution du processus en cours et en restaurer un autre, ce qui laisse l'unité de calcul inutilisée pendant ce temps. Sachant que ce contexte comprend l'ensemble des registres, pour utiliser l'unité de calcul d'une façon optimale, on gère plusieurs contextes simultanément, cela en dupliquant les registres du même cœur. Cette technique est appelée Multi-Threading (Hyper-Threading chez Intel), dans ce cas, un cœur physique apparaît comme deux (ou plus) cœurs logiques au système d'exploitation[6].
Le premier processeur Intel qui a intégré cette technologie était le processeur Intel Pentium 4.

I.6.4 Réseaux d'interconnexion

Avoir plusieurs cœurs sur une même puce requiert des mécanismes de communication entre les cœurs. Les premiers processeurs multi cœurs communiquait a travers un bus de communications commun partagé entre les cœurs, ce qui facilite la mis en œuvre de la cohérence des caches. mais le fait que plusieurs cœurs partagent le même bus limite la vitesse de communication.

Les dernières architectures des multi cœurs utilisent une barre transversale(crossbar), plusieurs bus en anneau(multiple ring busses), une grille(mesh) pour interconnecter les cœurs[7].

La figure I.16 donne un aperçu les réseaux d'interconnexion les plus utilisés actuellement.

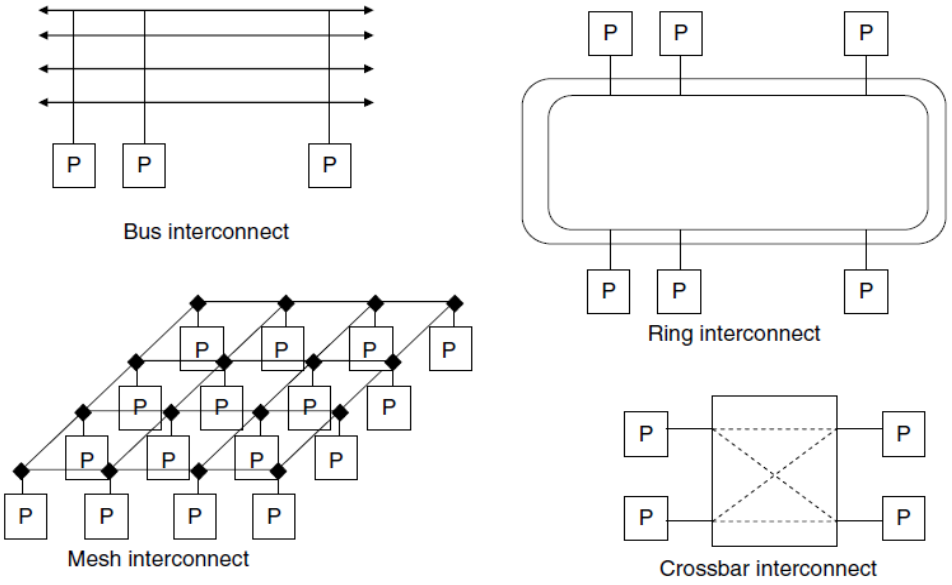


FIGURE I.16 – Réseaux d'interconnexion dans les multi cœurs

Comme le nombre de cœurs augmentera, les réseaux d'interconnexion seront de plus en plus face à des contraintes d'évolutivité et de puissance.
Les réseaux optiques sur puce sont au centre des efforts de recherche récents dans les universités et les entreprises de fabrication de processeurs tels que IBM ou Intel. En plus d'offrir des vitesses plus élevées, les interconnexions optiques ont une faible consommation électrique et occupant moins d'espaces, les interconnexions optiques ont la possibilité de couvrir plusieurs problèmes d'interconnexion[7].

I.6.5 Architecture multi cœurs homogène

C'est l'architecture de la plupart des processeurs multi cœurs, dans ce cas tous les cœurs du processeur sont identiques. Le majeur avantage de cette architecture est la facilité de la conception et la programmation des programmes, car on ne se soucie pas dans quel cœur s'exécute les tâches, et la répartition des tâches sur les cœurs peut se faire automatiquement.

I.6.6 Architecture multi cœurs hétérogène

Dans cette architecture le processeur comprend des cœurs différents qui peuvent varier à la fois dans le jeu d'instructions et les performances. L'organisation typique

a un processeur du contrôle qui gère les activités d'un ensemble de cœurs plus simple. Dans ce cas la conception et la réalisation des programmes sont plus compliquées par rapport au multi cœurs homogènes car chaque tâches doit s'exécuter sur un cœur particulier, et la répartition des tache doit se faire manuellement.

Le processeur CELL est l'un des exemples les plus récent de processeur multi-cœurs hétérogène. Le Cell est composé d'un cœur principal qui est un processeur PowerPC de IBM nommé PPE (Power Processor Element) et de huit cœurs auxiliaires qui sont des unités de calcul accélératrices appelées SPE (Synergestic Processor Element).

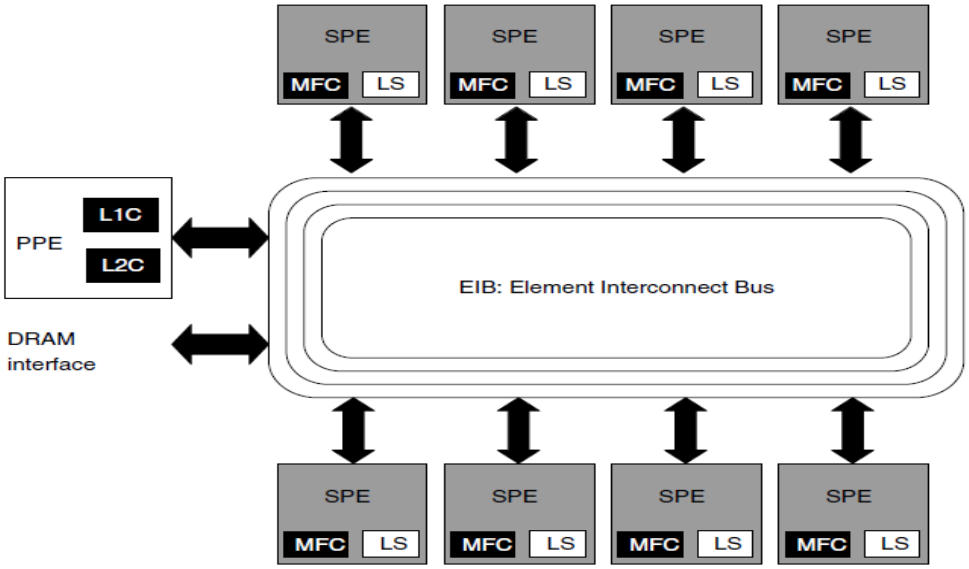


FIGURE I.17 – L'architecture hétérogène du processeur Cell

I.6.7 Type des caches dans les multicœurs

Il existe deux type de mémoire cache :

Mémoire caches partagée : la mémoire cache est partagée par tous les cœurs, qui peuvent y accéder de façon concurrente.

Mémoire caches dédiée : chaque cœur possède son propre cache, que lui seul peut utiliser. On trouve donc un cache pour chaque cœur.

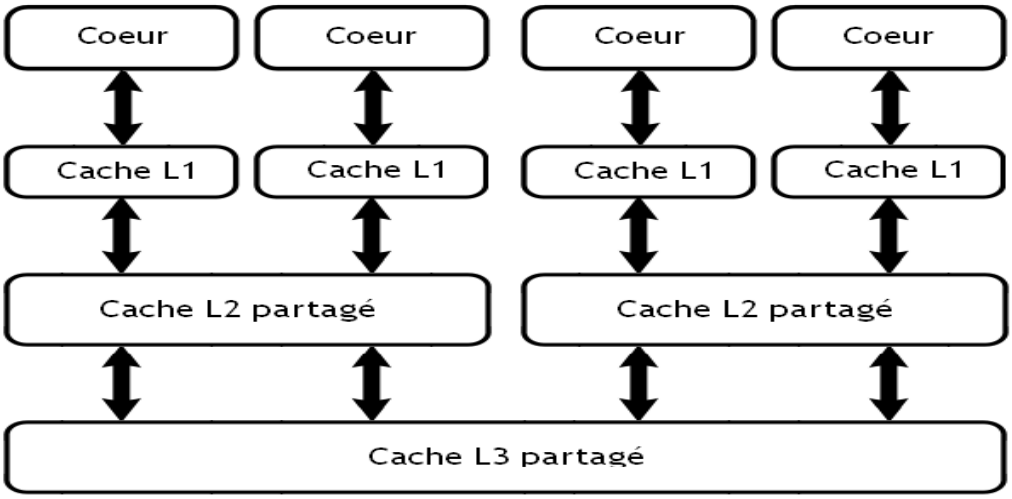


FIGURE I.18 – exemple d'une architecture a 4 cœurs avec un cache L1 dédié pour chaque cœur, L2 partagé entre deux cœurs et L3 partagé entre les quatre cœurs

I.6.8 La cohérence des caches dans les multi cœurs

Dans le cas où deux cœurs ou plus lisent une même donnée dans la mémoire centrale, une copie sera alors dans le cache de chaque cœur. Si l'un des cœurs modifie la donnée, la donnée sera modifiée sur son cache et sur la mémoire centrale, mais pas sur les caches des autres cœurs. Pour garantir la cohérence des données, cette modification doit être visible pour les autres cœurs, en d'autres termes les autres cœurs doivent être avertis de ce changement. Pour cela un protocole de cohérence des caches est utilisé.

I.6.9 Les processeurs "Many Integrated Core : MIC"

Comme nous l'avons vu, l'architecture des nouveaux processeurs a tendance à l'augmentation du nombre de cœurs. Après les multi-cœur présentés, les fabricants réalisent des processeurs many-core avec plusieurs dizaines de cœurs.

Exemple : Intel Xeon Phi 7100 (2013) avec 61 cœurs et deux niveaux de cache(L2 30.5MB).

I.6.10 Les processeurs multi cœurs Intel

La première génération des processeurs Intel Core est apparue en 2008. Depuis, les Intel Core se sont considérablement améliorés (de nouvelles technologies, de meilleures performances, des graphismes largement optimisés...). Aujourd'hui en 2016, on est à la sixième génération.

La principale amélioration de ces générations est la diminution de la finesse de gravure de 45nm de la 1ére génération à 14nm de 6éme génération.

Géneration	Nom	Finesse de gravure	année de sortie
1	Nehalem	45nm	Novembre 2008
2	Sandy Bridge	32nm	Janvier 2011
3	Ivy Bridge	22nm	Avril 2012
4	Haswell	22nm	Juin 2013
5	Broadwell	14nm	Septembre 2014
6	Skylake	14nm	Septembre 2015

TABLE I.3 – Générations des processeurs multi cœurs Intel

Diminuer la finesse de gravure permet de diminuer la quantité de chaleur produite par le processeur, sa consommation d'énergie et de pouvoir le monter plus haut en fréquence. Cela permet également de baisser les coûts de fabrication, ce qui est utile pour le fabricant.

a) L'architectures Ivy Bridge[8] :

Ivy Bridge est l'architecture de la troisième génération des processeurs multi cœurs d'Intel, elle succède à l'architectures Sandy Bridge, par rapport à laquelle elle apporte un grand changement dans la finesse de gravure, car cette génération utilise une nouvelle technologie de transistors qui est 3D Tri-gate Transistors.

On trouve des processeurs Ivy Bridge dans les gammes core i3, Core i5, Core i7 et Xeon. Le premier processeur Ivy Bridge a été commercialisé en 2012.

Les principales caractéristiques des processeurs de cette génération sont :

- Contrôleur mémoire intégré IMC(Integrated Memory Controller).
- Carte graphique intégrée (à partir de la deuxième génération).
- Interconnexion en ring (à partir de la deuxième génération).

- Support pour le Turbo Boost [6] d'Intel et le Hyper-Threading(Multi-Threading).
- Première utilisation des transistors tri-gate 3D(3D Tri-gate Transistors).
- Elle comporte trois niveaux de mémoire cache dont deux dédiés à chaque cœur.

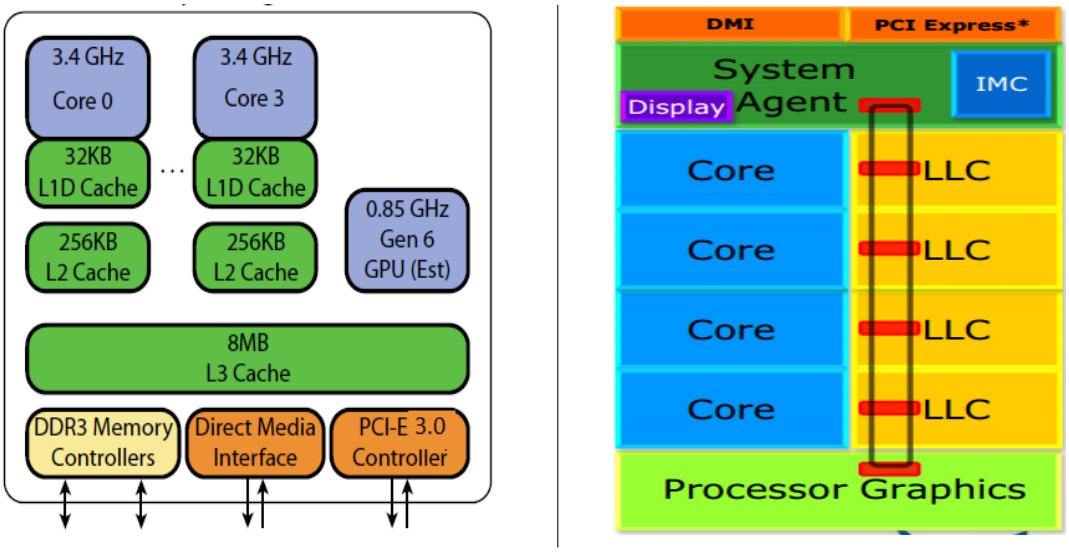


FIGURE I.19 – L'architecture Ivy Bridge de Intel

-IMC(Integrated Memory Controller) : gére les transactions avec la mémoire principale.

-DMI(Direct Media Interface) : Il est utilisé pour l'interconnexion entre le processeur et la PCH [7].

-PCI(Peripheral Component Interconnect) Express : interface qui gère les E/S, utilisée pour les cartes d'extension hauts débits comme carte vidéo.

-LLC(Last Level Cache) : c'est la mémoire cache de niveau 3.

les composants : DMI, contrôleur PCI, et IMC combinés forment l'Agent du Système(System Agent), qui est relié avec une interconnexion en ring au reste des composants : cœurs, LLC, processeur graphique[9].

Les générations qui succèdent a Ivy Bridge apportent des amélioration dans :

6. La technologie de Turbo Boost permet aux cœurs de fonctionner avec une fréquence plus vite que la fréquence du fonctionnement de base, si l'énergie total et la température l'autorisent.
7. puce qui gère les connexions SATA, USB, Ethernet

- la finesse de gravure, diminuée jusqu'à 14nm dans la 6éme génération.
- la vitesse de lecture et d'écriture dans la mémoire cache.
- comporte un processeur graphique plus performant.
- faible consommation d'énergie.

Et gardent l'interconnexion en ring et les trois niveaux de mémoire cache.

I.7 Conclusion

Nous avons vu dans ce chapitre les notions de base de l'architecture des premiers ordinateurs et les moyens utilisés pour augmenter leurs performances, Puis nous avons présenté la classification des machines parallèles et leur architecture mémoire, ainsi que les modèles de programmation associés à ces architectures mémoire et les concepts de base sur le parallélisme, avec les types de parallélisme, ses limites et comment évaluer les performances du parallélisme.

Nous avons vu aussi les bases de l'architecture des processeurs multi cœurs, en commençant par la raison de leur apparition (limites des processeurs mono cœur : chaleur, énergie, etc...), puis leur types (homogènes et hétérogènes), ainsi que les types de mémoires caches existant dans ces processeurs(partagé et dédié) et leurs avenir avec les MIC (Many Integrated Core). Enfin on a présenté les générations des processeurs multi cœurs de Intel, ainsi que les principales caractéristiques de la troisième génération.

Chapitre II

Intel TBB (Threading Building Blocks)

II.1 Introduction

Les processeurs multi cœurs deviennent de plus en plus disponibles sur le marché et utilisés par le grand public, et les processeurs mono cœurs deviennent relativement très rares. Ce qui fait de la programmation parallèles un grand intérêt. Mais écrire des programmes parallèles efficaces et scalables avec les threads traditionnels est très difficile, car le programmeur doit contrôler tous les détails de la programmation (la création des taches, la synchronisation, la répartition des charges, les communications, etc...).

Intel TBB est une librairie développée par Intel basée sur la STL (Standard Template Library) C++. Elle permet d'abstraire au maximum les détails complexes de la programmation parallèle, et aide à créer des applications qui tirent profit des avantages des processeurs multi cœurs.
Intel TBB est disponible sous double licence, une licence commerciale (299$ avec des support technique) et licence libre (GNU GPL [1], même version sans support technique).

Dans ce chapitre nous présenterons les avantages de Intel TBB, une description de ses composants et des exemples d'utilisations de quelques-uns de ces composants.

1. (General Public License) est une licence qui fixe les conditions légales de distribution des logiciels libres du projet GNU.

II.2 Les avantages de Intel TBB

- Intel TBB peut être utilisé avec tous les compilateur C++ (Microsoft, Intel, GNU.), et les systèmes d'exploitation Windows, LINUX, Mac OS.
- Intel TBB utilise les templates c++ permettant aux programmeurs de programmer des applications parallèles sans avoir à être des experts dans la synchronisation, l'équilibrage de charge, et l'optimisation de la mémoire cache.
- Intel TBB rend l'écriture des applications sacalables beaucoup plus facile qu'elle ne l'est avec les threads traditionnels.
- contrairement à Pthread, Intel TBB abstrait la création et la gestion des threads et offre des templates et containers qui facilitent la programmation parallèle.
- Intel TBB est compatible avec d'autres librairies telles que OpenMP et Pthread. Donc on n'est pas obligé d'utiliser seulement Intel TBB, mais on peut utiliser d'autre librairie avec.
- Intel TBB permet de créer des applications parallèles fiables, portables et évolutives.
- Intel TBB implémente une stratégie d'ordonnancement par vol de travail (le workstealing) pour l'équilibrage des charges.

II.3 Le Work-stealing(vole de travail)

Intel TBB utilise la notion de work-stealing pour distribuer les taches aux threads worker. Par défaut Intel TBB crée un thread worker par thread hardware, chaque thread worker maintient les deux extrémités de sa liste des taches. Quand une tache est engendrée, elle sera mise dans la queue de la liste de taches d'un thread worker. Quand un thread worker finit d'exécuter une tache il prend une autre tâche de la queue de la liste.
Si un thread worker ne trouve pas de taches dans sa liste, il prend une tache de la tête d'une liste d'un autre thread worker au hasard, ce qu'on appelle le work-stealing.
Puisque les taches peuvent s'exécuter dans n'importe quel thread worker, Intel TBB peut créer autant de threads qu'il veut selon la disponibilité des ressources.
La figure II.1 illustre la notion du work stealing.

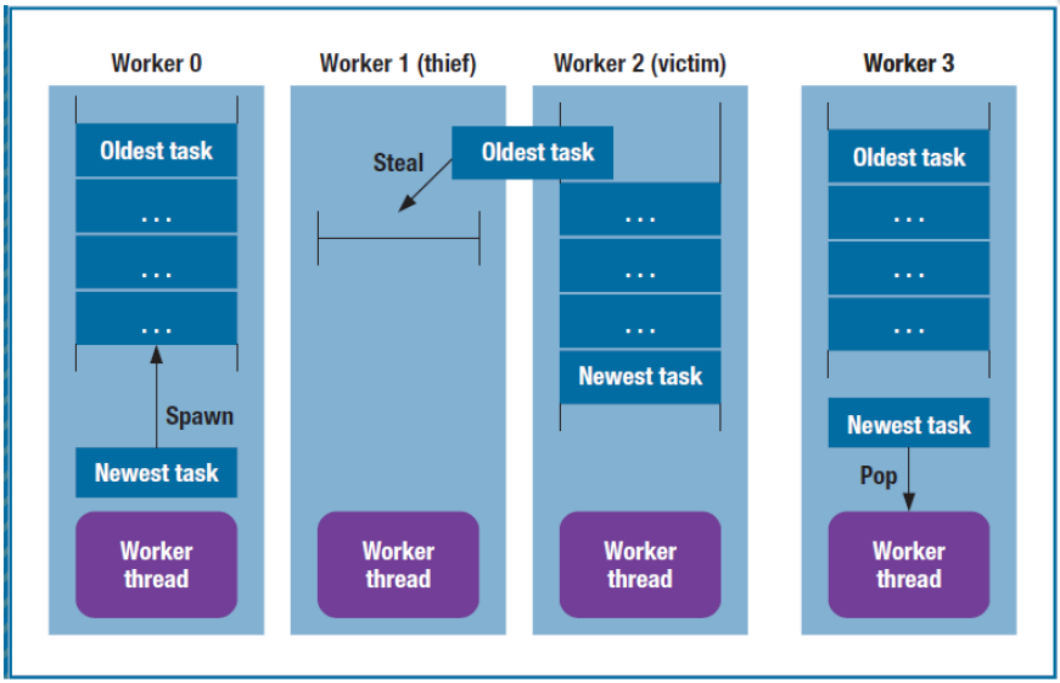


FIGURE II.1 – le work stealing

II.4 Les composant de Intel TBB

Pour aider le développeur dans la construction des programmes parallèles, Intel TBB fourni principalement les blocs suivants :

- Les algorithmes parallèles génériques.
- L'ordonnanceur de tâches
- Les primitives de synchronisation.
- Les containers concurrents.
- Les alocateurs mémoire.

La figure II.2 montre les composants (blocs) fournis par Intel TBB :

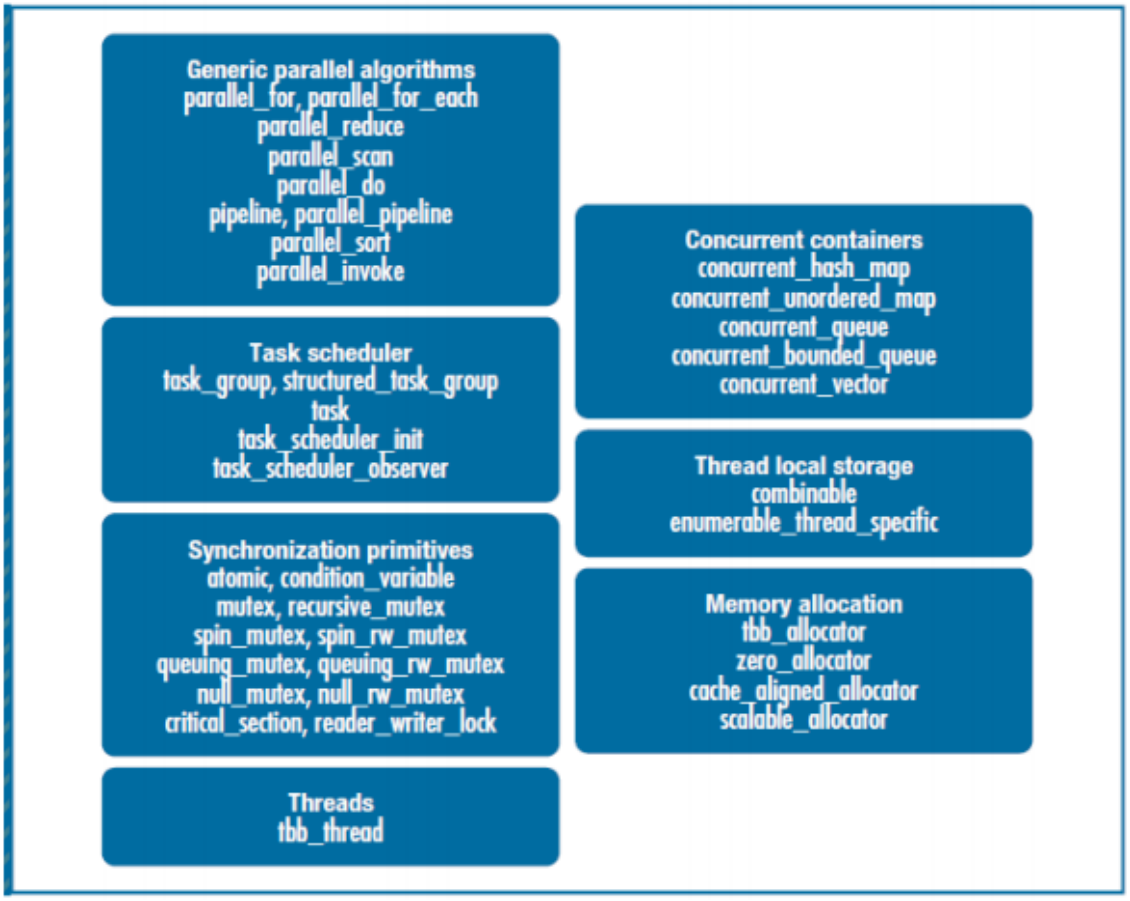


FIGURE II.2 – Les composant de Intel TBB.

II.5 Algorithmes de base

Les composants de Intel TBB sont définis dans le **namespace tbb**.

II.5.1 Initialisation de la librairie

Toute utilisation de Intel TBB doit être précédée de l'initialisation d'un objet de la classe **task_scheduler_init** qui est l'ordonnanceur de tâches (task scheduler). Par défaut, le constructeur de **task_scheduler_init** fait l'initialisation et le destructeur fait la terminaison. Donc il suffit de déclarer un objet **task_scheduler_init** dans la fonction principale **main()**.

```
1  #include "tbb/task_scheduler_init.h"
2  using namespace tbb;
3  int main( ) {
4          task_scheduler_init init;
5          ...
6          return 0;
7  }
```

Le constructeur de **task_scheduler_init** prend un paramètre facultatif qui spécifie le nombre de thread désirés, y compris le thread qui fait l'appel. Le paramètre facultatif peut être :

- La valeur **task_scheduler_init::automatic**, qui est aussi la valeur par défaut quand le paramètre n'est pas spécifié.
- La valeur **task_scheduler_init::deferred** qui diffère l'initialisation jusqu'à ce que **task_scheduler_init::initialize(n)** soit appelé. La valeur n peut être toute valeur légale pour le paramètre facultatif du constructeur..
- Un entier positif qui spécifie le nombre de thread à utiliser.

Le paramètre devrait être spécifié seulement pour l'étude de la scalabilité pendant le développement. Pour le code de la production, il vaut mieux ne pas spécifier le paramètre, ou utilisez **task_scheduler_init::automatic** pour que le code puisse s'adapter sur des processeurs avec un nombre de cœur différent.

II.5.2 parallele_for

la fonction **parallel_for** décompose un espace d'itération en blocs, et exécute chaque bloc sur un thread séparé. Le premier pas dans la parallélisation d'une boucle est de convertir le corps de la boucle sous forme d'une classe qui définit la méthode **opérator()** qui va opérer sur chaque bloc.
Le code suivant montre comment déclarer cette classe et l'utilisation de **parallel_for** :

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
class BoucleParallel {
        public:
                void operator( )( const blocked_range<size_t>& r ) const {

                        for( size_t i=r.begin(); i!=r.end( ); ++i )
                                //instructions
                }
};
void Parallel(size_t n ) {
        BoucleParallel boucleParallel;
        parallel_for(blocked_range<size_t>(0,n,GrainSize), boucleParallel );
}
```

blocked_range <T> est une classe template fournie par la bibliothèque qui décrit un espace de d'itération unidimensionnel.
La forme générale du son constructeur est **blocked_range <T>(begin,end,grainsize)**. Les paramétrés begin et end spécifient le début et la fin de l'espace d'itération. Le **grainsize**, spécifie le nombre d'itérations pour chaque blocs.
Si on a un tableau de 1000 case :

si grainsize=500 alors on aura 2 blocs.
si grainsize=200 alors on aura 5 blocs.
si grainsize=100 alors on aura 10 blocs.

La création des threads engendre un temps de latence pour chaque thread. Si les blocs sont trop petits, le temps de latence peut excéder le travail utile.

a) auto_partitioner

Un **partitioner** est un objet qui guide le partitionnement de l'espace d'itération. Quand le **grainsize** n'est pas spécifié, un **partitioner** doit être spécifié. Si le **grainsize** et le **partitioner** ne sont pas spécifiés, alors un **grainsize** de 1 sera spécifier automatiquement.

```
#include "tbb/parallel_for.h"
void Parallel( float a[], size_t n ) {
        BoucleParallel boucleParallel;
        parallel_for(blocked_range<size_t>(0,n), boucleParallel, auto_partitioner() );
}
```

l'**auto_partitioner** permet de faire un partitionnement dynamique par rapport au nombres de cœurs disponibles sur le processeur sur lequel s'exécute le programme.

II.5.3 parallele_reduce

L'application d'une fonction telle que la somme, max et min sur tous les éléments d'un tableaux est appelée une opération de réduction. Pour la parallélisation de ces fonctions Intel TBB offre la fonction **parallel_reduce**.
Le code suivant est le code séquentiel pour calculer la somme des éléments d'un tableau :

```
1  float Sequentiel( float a[], size_t n ){
2    float somme=0;
3          for( size_t i=0; i<n; ++i )
4                  somme+=f(a[i]);
5          return somme;
6  }
```

Intel TBB définit **parallel_reduce** semblable à **parallel_for**. La réduction précédente peut être parallélisée en utilisant **parallel_reduce** comme le montre le code suivant :

```
1  class Somme {
2          float* my_a;
3          public:
4                  float s;
5                  void operator( )( const blocked_range<size_t>& r ) {
6                          for( size_t i=r.begin(); i!=r.end( ); ++i )
7                                  s+=my_a[i];
8                  }
9                  Somme( Somme& x, split ) : my_a(x.my_a), s(0) {}
10                 void join( const Somme& y ) {somme+=y.s;}
11                 Somme(float a[] ) : my_a(a), s(0)
12                 {}
13 };
14 float ParallelSumFoo( const float a[], size_t n ) {
15         Somme sf(a);
16         parallel_reduce(blocked_range<size_t>(0,n),sf,auto_partitioner());
17         return sf.s;
18 }
```

Le constructeur de division prend comme arguments une référence à l'objet d'origine et à un argument de type split qui sert à distinguer le constructeur de division du constructeur de copie. Cet argument factice sert, par sa présence à distinguer les deux constructeurs.

La méthode **join** est appelée à chaque fois qu'une tâche finit son travail et a besoin de retourner le résultat au corps principal de la boucle.

II.5.4 Le pipeline

Le pipelining est un modèle parallèle qui imite une chaîne industrielle, le flux de données passe par les étages de pipeline, et dans chaque étage les données sont traitées différemment. Les classes **tbb::pipeline** et **tbb::filter** implémentent le modèle de piplining. Cette implémentation facilite la mise en œuvre de pipelining, parce que on aura pas besoin de créer les tâches, faire la synchronisation et passer les données d'un étage à un autre.
L'interface **tbb::filter** permet de définir les étages du pipeline, la méthode **operator** doit contenir le traitement a faire dans cet étage du pipeline.
La methode **add_filter(filter f)** de la classe **tbb::pipeline** permet d'ajouter les filtres au pipeline.

II.6 Création des tâches

Intel TBB fournit deux interfaces pour utiliser les taches : **task-group** et **task**.

a) task_group

La classe **task_group** permet de créer facilement des groupes de tâches parallèles. Elle définie une méthode **run**, dont chaque appel crée une tâche exécutant ce que lui est passé en argument.
L'exemple ci-dessous montre une implémentation récursive du calcul du nombre de Fibonacci en utilisant **task_group**. Cette mise en œuvre peut reproduire de nombreuses tâches en parallèles, et ne constitue pas une solution évolutive, elle doit être considérée comme un exemple uniquement syntaxique.

```
1  #include "tbb/task_group.h"
2  using namespace tbb;
3  int Fib(int n) {
4          if( n<2 ){
5                  return n;
6          }else{
7                  int x, y;
8                  task_group g;
9                  g.run([&]{x=Fib(n-1);}); // lancer une tache.
10                 g.run([&]{y=Fib(n-2);}); // lancer une autre tache.
11                 g.wait(); // attendre que les deux taches soient terminées.
12                 return x+y;
13         }
14 }
```

b) task

La classe **task**, permet un contrôle plus précis, mais pas aussi facile à utiliser que **task_group**. Chaque déclaration de tâche doit redéfinir la méthode virtuelle **task::execute** qui doit contenir le travail de la tâche et renvoyer soit NULL soit un pointeur vers la prochaine tâche à exécuter.
L'exemple ci-dessous montre une implémentation récursive du calcul du nombre de Fibonacci en utilisant la classe **task**.

```
1  const int CutOff = 16;
2  class FibTask: public task {
3  public:
4          const long n;
5          long* const sum;
6          FibTask( long n_, long* sum_ ):n(n_), sum(sum_)
7          {}
8          task* execute( ) { // Overrides virtual function task::execute
9                  if( n < CutOff ) { //Vérifie si n est si petite (l'exécution séquentiel serait plus rapide).
10                 *sum = SerialFib(n);
11                 }else {
12                         long x, y;
13                         FibTask& a = *new(allocate_child( )) FibTask(n-1,&x);
14                         FibTask& b = *new(allocate_child( )) FibTask(n-2,&y);
15                         set_ref_count(3);// Set ref_count to "two children plus one for the wait".
16                         spawn( b );// Lancer l'execution de b.
17                         spawn_and_wait_for_all(a);//Lancer l'execution de a et attendre tous les fils (a
                               and b).
18                         *sum = x+y;
19                 }
20                 return NULL;
21         }
22 };
23 long ParallelFib( long n ){
24         long sum;
25         FibTask& a = *new(task::allocate_root( )) FibTask(n,&sum);
26         task::spawn_root_and_wait(a);
27         return sum;
28 }
```

II.7 Les containers concurrents

Les containers concurrents sont très importants parce que les containers STL ne sont pas concurrents, et essayer de les modifier concurremment peut corrompre les containers. En conséquence, les containers STL sont protégés, en laissant seulement un thread opérer à la fois sur le container. Ce qui n'est pas favorable pour le paral-

lélisme. Intel TBB offre des container concurrents qui autorisent plusieurs threads à opérer simultanément sur le même container.

a) concurrent_queue

la classe **concurrent_queue<T>** implémente une file (FIFO) concurrente avec des valeurs de type **T**, et plusieurs threads peuvent insérer et récupérer des éléments de la file.
Les méthodes fondamentales de cette classe sont : **push()** et **try_pop()**, la méthode push permet l'insertion dans la file et la méthode try-pop() récupère un item de la file.

b) concurrent_vector

La classe **concurrent_vector<T>** est un vecteur dynamique d'éléments de type **T** qui permet l'accès concurrent à ses elements.
cette classe a deux méthodes pour le redimensionner : **grow_by** et **grow_to_at_least**. La méthode **grow_by(n)** permet d'attacher **n** éléments consécutifs à un vecteur, et retourne l'index du premier élément attaché. La méthode **grow_to_at_least(n)** permet de redimensionner la taille du vecteur à la taille **n**.

c) concurrent_hash_map

La classe **concurrent_hash_map<Key,T,HashCompare>** est une table de hachage qui permet l'accès concurrent. La table est une carte d'une clef à un type **T**. **HashCompare** doit être une classe ou structure qui définit deux méthode :

- **bool HashCompare::equal(const Key& j, const Key& k) const** : définit comment hacher une clef.
- **size_t HashCompare::hash(const Key& k)** : définit comment comparer deux clefs.

La classe **concurrent_hash_map** définit les méthodes suivantes :

- **insert(accessor& result, const Key& key)** : permet de d'insérer un nouvelle paire dans la table .
- **find(accessor& result, const Key& key)** : permet de chercher dans la table une paire avec la clé donnée. Si la clé est trouvé, fournir un accès en écriture à la paire correspondante.
- **erase(const Key& key)** : permet de chercher dans table une paire avec la clé donnée. Supprime la paire correspondante si elle existe.

accessor est un attribut de **concurrent_hash_map** qui est pointeur sur la table elle même.
l'exemple suivant montre comment on peut utiliser **concurrent_hash_map** :

```
1  #include "tbb/concurrent_hash_map.h"
2  #include "tbb/blocked_range.h"
3  #include "tbb/parallel_for.h"
4  #include <string>
5  using namespace tbb;
6  using namespace std;
7  struct MyHashCompare {
8      static size_t hash( const string& x ) {
9          size_t h = 0;
10         for( const char* s = x.c_str(); *s; ++s )
11             h = (h*17)^*s;
12         return h;
13     }
14     static bool equal( const string& x, const string& y ) {
15         return x==y;
16     }
17 };
18 typedef concurrent_hash_map<string,int,MyHashCompare> StringTable;
19 struct Tally {
20     StringTable& table;
21     Tally( StringTable& table_ ) : table(table_) {}
22     void operator()( const blocked_range<string*> range ) const {
23         for( string* p=range.begin(); p!=range.end(); p++ ) {
24             StringTable::accessor a;
25             table.insert( a, *p );
26             a->second += 1;
27         }
28     }
29 };
30 const size_t N=1000000;
31 string Data[N];
32 void CountOccurrences() {
33     StringTable table;
34     parallel_for( blocked_range<string*>( Data, Data+N, 1000 ),
35                   Tally(table) );
36 }
```

II.8 Les primitives de synchronisation

Pour la synchrinisation Intel TBB offre les mutexes et Les opérations atomiques.

II.8.1 Les mutexes

Un **mutex** est un objet sur lequel un thread peut acquérir un verrou, et seulement un thread à la fois peut avoir un verrou sur le **mutex**, les autres threads doivent attendre que le **mutex** soit libéré.
Tous les **mutexes** dans Intel TBB ont une interface similaire et facile à apprendre. Par exemple, tous les **mutexes** ont un objet scoped_lock qui implémente les méthodes **acquire()** et **release()**.
Le plus simple des **mutex** offert par Intel TBB est **spin_mutex**, le code suivant montre comment utiliser le **spin_mutex**.

```
1    spin_mutex mutex;
2    spin_mutex::scoped_lock lock;
3    lock.acquire(mutex);
4
5    //instructions
6
7    lock.release();
```

II.8.2 Les opérations atomiques

Une opération atomique est une opération composée de plusieurs actions, qui se déroule sans pouvoir être interrompue par un autre thread.
La classe **atomic<T>** permet de déclarer une variable atomique de type **T** qui prend en charge des opérations atomiques.

Opérations	Descriptions
=x	Lire x.
x=	Ecriture de x et retourne sa valeur
x.fetch_and_store(y)	x=y , et retourne l'ancienne valeur de x.
x.fetch_and_add(y)	x+=y, et retourne l'ancienne valeur de x.
x.compare_and_swap(y,z)	Si x=z alors x=y, et retourne l'ancienne valeur de x.

TABLE II.1 – Les opérations atomiques fondamentales pour une variable atomique

II.9 Allocation de la mémoire

L'allocation de la mémoire devient un goulot d'étranglement dans un programme multi thread à cause de la concurrence des threads pour l'allocation de la mémoire.

A cause de cette concurrence, les programmes qui font usage intensif d'allocation de la mémoire peuvent être ralentis autant que le nombre des cœurs du processeur augmente.
Un autre problème pour les programmes parallèles est appelé faux partage. Le faux partage se produit lorsque plusieurs threads utilisent des emplacements de mémoire proches. Parce que les cœurs de processeur vont chercher et maintenir la mémoire dans les lignes de cache, tout accès à la mémoire au sein de la même ligne de cache doit être effectué uniquement par le même thread. Sinon, les accès à la mémoire sur la même ligne de cache vont provoquer un conflit et un remplacement inutile des lignes de cache, ce qui entraîne un grand ralentissements[10].

Comme solution aux problèmes de l'allocation concurrente, Intel TBB offre deux choix, semblables dans la façon de leur utilisation à l'allocateur mémoire standard STL std::allocator :

a) scalable_allocator

Ce modèle offre juste l'évolutivité, mais elle ne protège pas complètement contre le faux partage. La mémoire est retournée à chaque thread à partir d'un bassin séparé, ce qui aide à protéger contre le faux partage si la mémoire n'est pas partagée avec d'autres threads.
Pour remplacer les fonctions malloc, free, realloc et calloc :

```
void * scalable_malloc (size_t size) ;
void scalable_free (void* ptr) ;
void * scalable_realloc (void* ptr, size_t size) ;
void * scalable_calloc (size_t nobj, size_t size) ;
```

b) cache_aligned_allocator

Ce modèle offre à la fois l'évolutivité et la protection contre les faux partages. Il aborde le faux partage en veillant que chaque attribution se fait sur une ligne de cache.

scalable_allocator et **cache_aligned_allocator** sont identiques à **std::allocator**(allocateur standard), donc ils peuvent être utilisés de la même façon. Le code suivant montre

comment déclarer un vecteur STL qui utilise **cache_aligned_allocator** pour l'allocation :

```
std::vector< int,cache_aligned_allocator<int> >;
```

II.10 Expérimentation de Intel TBB

a) Le programme

Le programme suivant est un exemple d'utilisation de Intel TBB. Le programme utilise la fonction template **parallel_for** pour faire un calcul sur une matrice(4000x4000).

```
1  #include <iostream>
2  #include <cmath>
3  #include <tbb/tbb.h>
4  using namespace std;
5
6  const int taille = 4000;
7  double input[taille][taille];
8  double output[taille][taille];
9
10 class ApplyFoo {
11 public:
12         void operator()(const tbb::blocked_range<int>& r) const {
13                 float a;
14                 for (int i = r.begin(); i != r.end(); ++i)
15                         for (int j =0; j<taille; j++)
16                                 a =sqrt(pow(cos(input[i][j])*sin(input[i][j]),4))*sqrt(pow(cos(input[i][j])*sin(
                                        input[i][j]),4));
17                                 a=sqrt(pow(cos(a)*sin(a),15))*sqrt(pow(cos(a)*sin(a),15));
18                                 a=sqrt(cos(a)*sin(a))*sqrt(cos(a)*sin(a));
19                                 output[i][j]=sqrt(pow(cos(a)*sin(a),6))*sqrt(pow(cos(a)*sin(a),15));
20         }
21 };
22 int main(int argc,char * argv[])
23 {
24         tbb::task_scheduler_init init;
25         for (int i = 0;i < taille;i++)
26                         input[i] = i;
27         tbb::tick_count start,end;
28         std::cout<< "debut du tratement parallele" << std::endl;
29         start=tbb::tick_count::now();
30         tbb::parallel_for(tbb::blocked_range<int>(0,taille), ApplyFoo(),tbb::auto_partitioner());
31         end=tbb::tick_count::now();
32         std::cout << "fin du tratement parallele " << (end-start).seconds() << std::endl;
33         system("pause");
34         return 0;
35 }
```

Les figures II.3 et II.4 montrent respectivement l'utilisation du processeur lors de l'exécution de programme séquentiel et lors de l'exécution de programme parallèle.

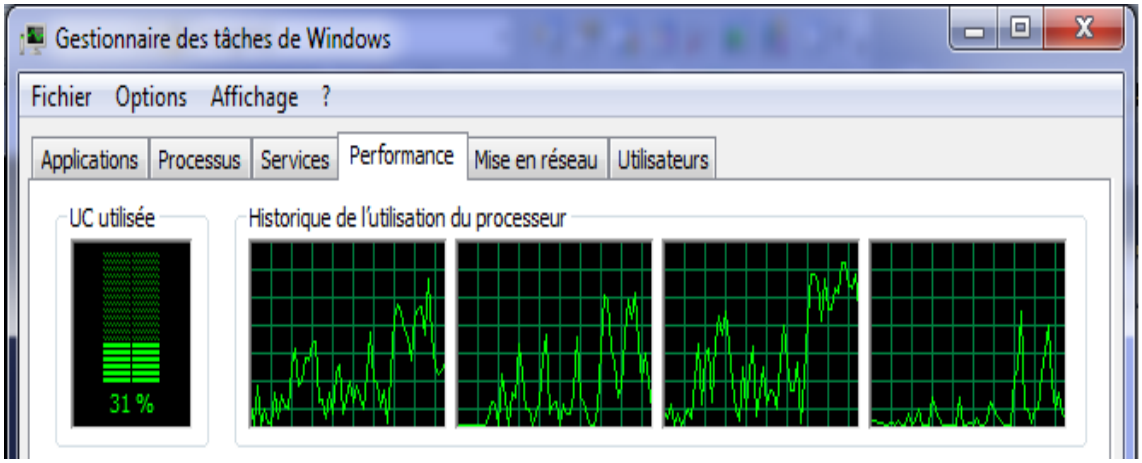


FIGURE II.3 – Utilisation du processeur lors de l'exécution du programme séquentiel

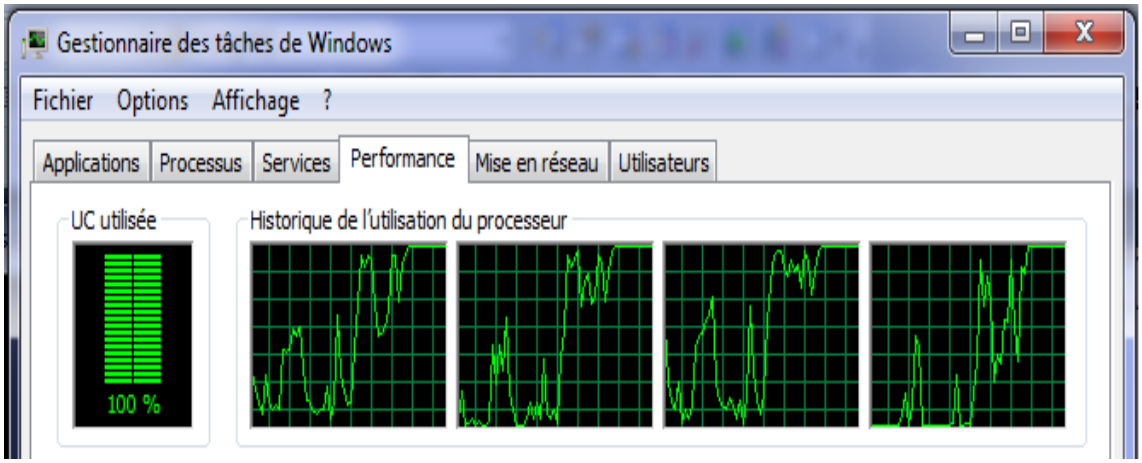


FIGURE II.4 – Utilisation du processeur lors de l'exécution du programme parallèle

On constate sur les figures précédente que lors de l'exécution de programme séquentiel le processeur n'est utilisé qu'à 31% de ses capacités, par contre lors de l'exécution parallèle le processeur est utilisé a 100% de ses capacités.

b) Les résultats

La table II.2 montre les résultats d'exécution du programme précédent généré par un compilateur Microsoft sur un processeur Intel i5 de 3ème génération avec 2

cœurs physiques, 4 cœurs logiques(2 threads pour chaque cœur physique).
On calcule l'accélération et de l'efficacité avec les formules suivantes :
Accélération=temps d'exécution séquentiel/temps d'exécution parallèle.
Efficacité= Accélération/ nombre de processeurs.

Algorithme	Temps d'execution(seconde)	Acceleration	Efficacité
séquentiel(1 thread)	17.50	1.00	1.00
parallèle (2 threads)	9.39	1.86	0.93
parallèle (3 threads)	6.82	2.56	0.85
parallèle (4 threads)	5.54	3.15	0.78

TABLE II.2 – résultats de l'expérimentation(Temps d'exécution, accélération et efficacité)

II.11 Conclusion

Dans ce chapitre nous avons présenté la librairie Intel TBB, ses avantages, ses composants, ainsi qu'une description de ces composants les plus pertinents. On a vu comment cette librairie facilite le développement d'applications parallèles en offrant des fonctions telles que **parallel_for**, **parallel_reduce** etc... . On termine cette présentation par une expérimentation de cette librairie avec l'implémentation d'un exemple de calcul sur une matrice(4000x4000), calculant l'accélération et l'efficacité de calcul parallèle effectué par Intel TBB.
Intel TBB est une librairie de parallélisation de plus en plus populaire qui encourage la création d'applications parallèles portables et scalables grâce aux composants qu'elle offre.

Chapitre III

Traitement d'image et extraction de contours

III.1 Introduction

Le traitement d'image permet de soustraire des informations à partir des caractéristiques d'une image. De nombreux problèmes d'application nécessitent un traitement préalable de l'image afin d'en détecter les contours. Les contours constituent en effet des indices riches pour certaines interprétations. En astrologie par exemple, on désire à partir d'un cliché pris par un télescope, pouvoir détecter de nouveaux corps célestes. Par ailleurs, en imagerie médicale, il est important de pouvoir déceler des tumeurs cancéreuses, à partir d'une scanographie. Dans l'exemple qui suit, nous présenterons une des nombreuses méthodes de détection utilisées dans l'imagerie médicale et satellite et dans la reconnaissance faciale aussi.

III.2 Concepts généraux

III.2.1 Définition de l'image

L'image est une représentation d'une personne ou d'un objet par la peinture, la sculpture, le dessin, la photographie, le film, etc. C'est aussi un ensemble structuré d'informations qui, après affichage sur l'écran, ont une signification pour l'œil humain.

III.2.2 Images numériques

une image numérique est une matrice de pixels repérés par leur coordonnées (x,y). S'il s'agit d'une image en couleur, un pixel est codé par 3 composantes (r,g,b) (chacune comprise entre 0 et 255), représentant respectivement les "doses" de rouge, vert et bleu qui caractérisent la couleur du pixel. S'il s'agit d'une image en niveau de gris, elle est codée par 1 composante comprise entre 0 et 255, représentant la luminosité du pixel.

III.2.3 Traitements d'image

Pour manipuler une image, on travaille sur un tableau d'entiers qui contient les composantes de chaque pixel. Les traitements s'appliquent toujours aux images en niveau gris et parfois aussi sur des images couleur. Nous allons distinguer plusieurs types de traitements dans ce projet.

a) Définition de pixel

C'est le plus petit éléments dans l'image est l'unité indivisible permettant de coder l'information relative à la luminosité en une certaine position. Les pixels sont en général carrés. **Pixel** vient de mot anglais **picture element**, son ensemble forme une image. Dans le cas d'image noir et blanc le pixel est codé sur un octet et dans une image en couleur (rouge,vert,blue) il est codé sur trois octets, un octet pour une couleur[11]. Voici un exemple des pixels d'une image sur un plan à deux dimensions :

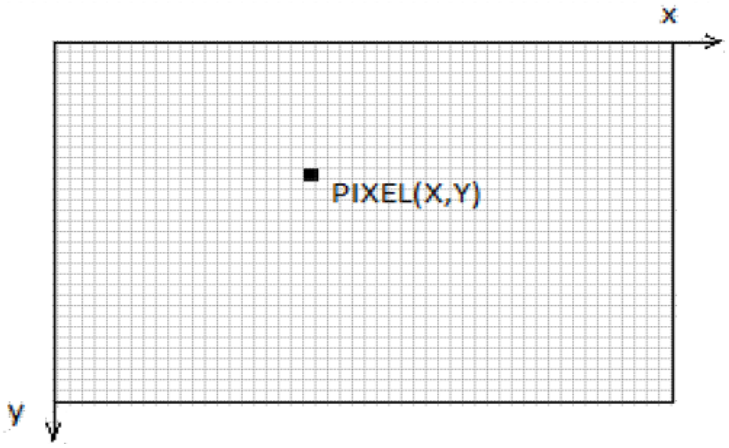


FIGURE III.1 – Pixel sur une image numirique

b) Voisinage d'un pixel

Le 4-voisinage d'un pixel de cordonner (x,y) est constitué de quatre pixels (x, y+1),(x, y-1),(x+1, y), (x-1, y). Le 8-voisinage d'un pixel de cordonner (x,y) est constitué de huit pixels (x,y+1),(x,y-1),(x+1,y),(x-1,y),(x+1,y+1),(x-1,y+1),(x+1,y-1),(x-1,y-1)[11].

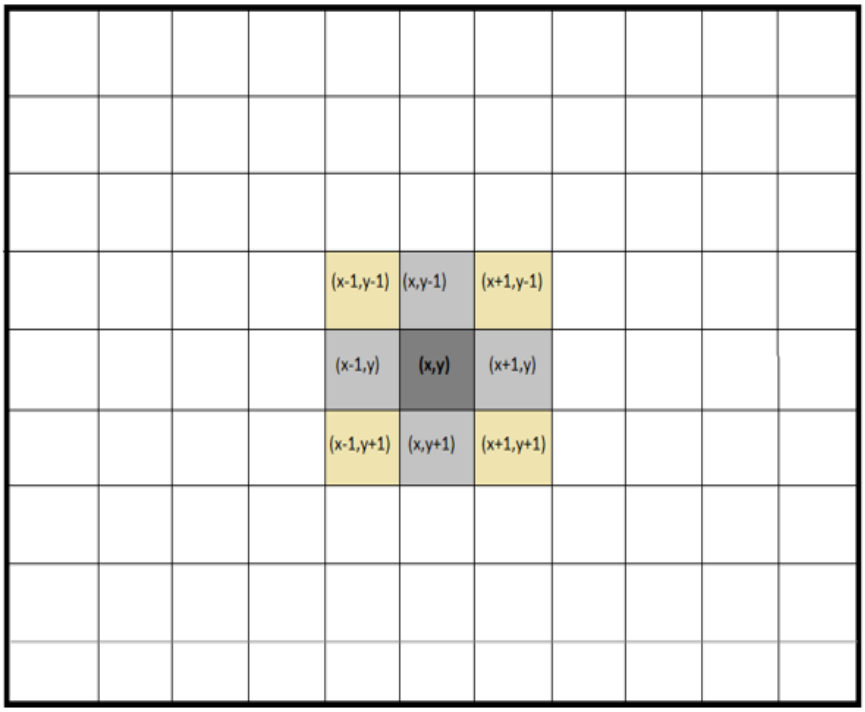


FIGURE III.2 – Voisinage d'un pixel

c) Résolution d'une image

La résolution indique le nombre de pixels au cm2 (ou pouce / inch : abrégé ppp ou dpi). Une résolution élevée augmente la taille des images. La résolution est la capacité à distinguer les détails fins dans l'espace. Une image destinée à un usage à l'écran (web) n'a pas besoin d'une grande résolution : 72 pixels /pouce suffisent. Les images destinées à être imprimées nécessitent une plus haute résolution : de 150 à 300 pixels/pouce.

d) Dimension de l'image

C'est la taille de l'image. Cette dernière se présente sous forme de matrice dont les éléments sont des valeurs numériques représentatives des intensités lumineuses (pixels). Le nombre de ligne de cette matrice multiplié par le nombre de colonnes nous donnes le nombre total de pixels.

III.2.4 Image en couleurs

Dans une image en couleur chaque pixel possède une couleur décrite par trois composantes correspondantes à la quantité de rouge (R), vert (G) et bleu (B)[12].

III.2.5 Image en niveau de gris

Le niveau de gris est la valeur d'intensité lumineuse en un point. La couleur du pixel peut prendre des valeurs allant du noir au blanc en passant par un nombre fini de niveaux intermédiaires. Donc pour représenter une image en niveau de gris, on peut attribuer à chaque pixel de l'image une valeur correspondante à l'intensité lumineuse de ce pixel. Cette valeur peut être comprise entre 0 et 255 (0 : le noir absolu, 255 : le blanc parfait). chaque pixel n'est donc plus représenter par un bit, mais par un octet[12].

a) La luminance

désigne le signal qui détermine les valeurs de contraste d'une image, du noir le plus profond jusqu'au blanc le plus pur. Elle est définie par la formule suivante[12] :

$$L = \frac{\alpha * R + \beta * V + \gamma * B}{\alpha + \beta + \gamma}$$

α, β, γ : étant des coefficients correspondant aux luminances des trois couleurs de base :

R(255,0,0) V(0,255,0) B(0,0,255)

$\alpha = 0.59, \beta = 0.30, \gamma = 0.11$.

III.2.6 Le bruit

Le bruit d'image est la présence d'informations parasites qui s'ajoutent de façon aléatoire aux détails de la scène photographiée numériquement. Il est plus particulièrement visible dans les zones peu éclairées, où le rapport signal/bruit est faible,

mais aussi dans les parties uniformes telles qu'un ciel bleu. Il a pour conséquence la perte de netteté dans les détails.

III.2.7 Le filtrage

Le filtre consiste à appliquer une transformation à toute ou une partie de l'image en appliquant un opérateur. Beaucoup de traitements d'images sont basés sur les produits de convolutions, ce dernier est un opérateur mathématique qu'on utilise pour multiplier des matrices entre elles.
Le principal objectif du filtrage est de prétraiter une image afin que l'image filtrée soit plus adaptée que l'image originale pour une application spécifique. Pour cela on multiplie chacun des pixels de la matrice correspondantes à l'image a traité par le noyau [k](matrice masque), pour calculer la valeur d'un pixel I(x, y) de la matrice image, on multiplie sa valeur par celle du pixel central du noyau K(2,2) et on additionne ensuite la valeur des produits des pixels adjacents.
Dans cet exemple On applique un produit de convolution au pixel central on obtient d'après le calcul $(150\times5)+(-1\times100)+(-1\times100)+(-1\times100)+(-1\times100)=350$.

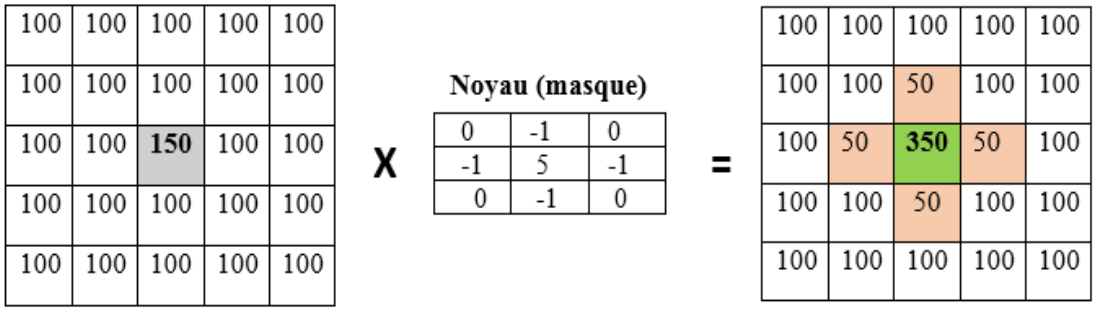


FIGURE III.3 – Exemple de filtrage

III.2.8 Type des filtres

a) Filtre passe-haut

Un filtre passe-haut est un filtre qui laisse passer les hautes fréquences et qui atténue les basses fréquences, c'est à dire les fréquences inférieures à la fréquence de coupure. Il pourrait également être appelé filtre coupe-bas.

b) Filtre passe-bas

Un filtre passe-bas est un filtre qui laisse passer les basses fréquences et qui atténue les hautes fréquences, c'est à dire les fréquences supérieures à la fréquence de coupure. Il pourrait également être appelé filtre coupe-haut. Le filtre passe-bas est l'inverse du filtre passe-haut.

c) Filtre passe-bande

Un filtre passe-bande est un filtre qui ne laisse passer qu'une bande ou intervalle de fréquences compris entre une fréquence de coupure basse et une fréquence de coupure haute du filtre. Le concept de filtre est une transformation mathématique appliquée à des données (un signal).

III.3 Extraction de contours

Le but de la détection de contours est de repérer les points d'une image numérique qui correspondent à un changement brutal de l'intensité lumineuse. Ces changements de propriétés de l'image traduisent en général des événements importants ou des changements dans les propriétés du monde. Ils incluent des discontinuités dans la profondeur, dans l'orientation d'une surface, dans les propriétés d'un matériau et dans l'éclairage d'une scène. La détection de contour est un champ de la recherche qui appartient au traitement d'image et à la vision par ordinateur, particulièrement dans le domaine de l'extraction de caractéristiques.
La détection des contours d'une image réduit de manière significative la quantité de données et élimine les informations qu'on peut juger moins pertinentes, tout en préservant les propriétés structurelles importantes de l'image.

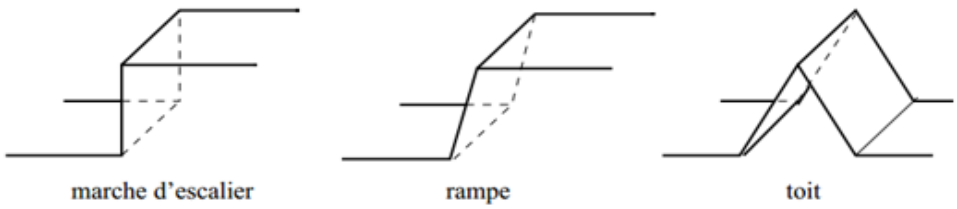


FIGURE III.4 – Quelques modèles de contours.

III.3.1 Approche pour l'extraction de contour

Il existe en traitement d'images différentes approches pour effectuer la détection de contour. Les méthodes les plus immédiates sont des méthodes dérivatives qui détectant les variations locales d'intensité des pixels d'une image. Les contours peuvent être obtenus soit par extraction des maxima locaux de la dérivée première, soit par extraction des passages par zéro de la dérivée seconde directionnelle. Les deux approches suivantes, appartiennent respectivement aux deux méthodes ci-dessous :

-approche de laplacien : détermine le passage à zéro du laplacien.

-approche de gradient : détermine les extrema locaux dans la direction du gradient.

a) Approche laplacien :

Le filtre Laplacien est un filtre de convolution particulier utilisé pour mettre en valeur les détails qui ont une variation rapide de luminosité. Le Laplacien est donc idéal pour rendre visible les contours des objets, d'où son utilisation dans la reconnaissance de formes dans des applications militaires, puis civiles. D'un point de vue mathématique, le Laplacien est une dérivé d'ordre 2 à deux dimensions. Le Laplacien d'une image d'intensité I(x,y) est défini par l'équation suivante :

$$L(x,y) = \frac{d^2I(x,y)}{dx^2} + \frac{d^2I(x,y)}{dy^2}$$

La détection s'effectue en deux étapes :

1. Détection des passages par zéros : les pixels pour lesquels le Laplacien change de signe sont sélectionnés.
2. Seuillage des passages par zéros de fortes amplitudes.

b) Approches Gradient :

Un gradient permet de visualiser les variations d'un phénomène, ainsi un dégradé de couleur peut s'appeler un gradient de couleur.
Mais il est parfois intéressant d'avoir à mettre en avant des détails qui changent selon une direction donnée. Dans ce cas on utilise des filtres de type gradient, qui sont en fait des dérivés partielles suivant une direction particulière, en général l'un des deux axes cartésiens X ou Y de l'image. L'extraction des points contours s'effectue par la sélection des points dont la norme de gradient est élevée, ceci s'effectue en deux étapes[13] :

1. calcule de la norme gradient .

$$N(x,y) = \sqrt{Ix(x,y)^2 + Iy(x,y)^2}$$

2. sélection des points dont la norme de gradient est supérieure à un seuil donné.

$$G(x,y) > s \qquad avecs : seuilfixeàpriori.$$

c) Le filtre de Canny

On se place dans le cas monodimensionnel. On suppose que la détection est effectuée en convoluant le signal par un filtre de réponse impulsionnel h, les contours étant caractérisé par les extrema de la sortie du filtre. Les contours envisagés ici sont des contours de types marche et le bruit est supposé blanc (de moyenne nulle). Les critères d'optimalités sont[14] :

- détection : le contour doit être détecté, il faut minimiser les fausses réponses.

- localisation : le contour doit être localisé avec précision, il s'agit de minimiser la distance entre les points détectés et le vrai contour.

- réponse unique : il s'agit de minimiser le nombre de réponse pour un seul contour.

L'optimisation de ces critères dans ce modèle conduit a la résolution d'une équation différentielle :

$$2f(x) - 2\lambda a_1 f'(x) + 2\lambda a_2 f''(x) + \lambda a_3 = 0$$

dont la solution est de la forme :

$$h(x) = a_1 e^{\alpha x} cos(\omega x) + a_2 e^{\alpha x} sin(\omega x) + a_3 e^{-\alpha x} cos(\omega x) + a_4 e^{-\alpha x} sin(\omega x)$$

Avec les conditions initiales suivantes :

$$f(0) = f(W) = 0 \qquad f''(0) = S \qquad f''(W) = 0$$

d) Approche Deriche :

Deriche a proposé un filtre de lissage dont la dérivée est la solution de l'équation de Canny étendue sous forme de filtre à réponse impulsionnelle, c'est la même équation différentielle que celle de canny, sauf que les conditions aux limites sont différentes [13] :

$$f(0) = 0 \qquad f''(0) = S$$
$$f(+\infty) = 0 \qquad f''(+\infty) = 0$$

III.4 Filtre de deriche

L'approche développée par Deriche consiste à rechercher l'opérateur optimal sous forme de filtre à réponse impulsionnelle infinie. Il en résulte une équation différentielle admettant comme solution l'opérateur suivant :

$$f(x) = Ce^{-\alpha x}sin(\omega x)$$

α est un paramètre (strictement positive) qui caractérise la largeur de transition d'une frontière entre deux zones, Plus α est grand, plus on localise précisément le contour. Plus il est petit, plus on détecte facilement la présence des contours[14].

$$C = \frac{(1 - e^{-\alpha|x|})^2}{e^{-\alpha}}$$

Dans le cas ou $(\omega << 1)sin(\omega x) \approx x$ d'où le filtre devient :

$$d(x) = cxe^{-\alpha|x|}$$

Grâce à l'ensemble de ces critères : détection, localisation, séparabilité, réponse unique, Récursivité, complexité et faible nombre d'opérations par pixel, nous avons choisie d'utiliser le filtre de Deriche, qui a l'avantage de minimiser l'erreur en position et maximiser le rapport Signal/bruit. Les étapes de ce filtre sont données par l'algorithme suivant :

III.4.1 Les étapes de filtre de deriche

a) Le lissage

Le lissage élimine les petits détails crées par le bruit. Pour cela Deriche utilise un filtre passe bas qui fait lisser l'image c'est-à-dire éliminer les détails de haute fréquence sur cette dernière.

Le filtre S de lissage est le suivant :

$$S(x) = a(\alpha|x| + 1)e^{-\alpha|x|} \quad Avec \quad a = \frac{(1 - e^{-\alpha|x|})^2}{(1 + \alpha e^{-\alpha} - e^{-2\alpha})}$$

b) Le gradient

Permet de ne retenir de l'image traitée que les zones de contraste élevé, c'est-à-dire ou l'intensité lumineuse passe de façon brutale, d'une valeur élevée a une valeur faible(ou vice versa).

Le calcul des composantes du gradient s'effectue en calculant la dérivée directionnelle selon x et le résultat d'un lissage suivant la direction y, suivi d'une dérivation suivant x [13].

Gradient en x : Ix(x,y) =(Im*l(y)) * d(x)

Gradient en y : Iy(x,y) =(Im*l(x)) * d(y)

Avec :
d(x) : filtre de dérivation selon x.
d(y) : filtre de dérivation selon y.
l(x) : filtre de lissage selon x. l(y) : filtre de lissage selon y.

c) Extraction des maxima locaux

L'extraction des maxima locaux se résume à comparer la norme du gradient d'un point donné avec celle de ces voisins, ce point est pris comme maxima local si sa norme est supérieure à celles de ces voisins.

d) Seuillage

Après extraction des maxima locaux, une autre opération s'applique à l'image : c'est le seuillage. Son rôle est de produire une image dont les pixels ne sont codés qu'en noir et le blanc. Le seuil est établi on appliquant une méthode statique qui consiste à calculer la moyenne des niveaux de gris des points des maximas locaux sans considérer le fond de l'image. Le seuil S est calculé par la formule suivante :

$$S = \frac{\sum_{i=0}^{l-1} \sum_{j=0}^{c-1} Ng(i,j)}{M}$$

Ng : le niveau de gris du point maxima local.
M : le nombre de points non nuls dans l'image maxima locaux.
l : le nombre de ligne de l'image.
c : le nombre de colonne de l'image.
NG(i,j) = 255 pour NG(i ,j) > S.
où NG(i,j) =0 pour NG (i ,j) < S.

III.5 Implémentation de filtre Deriche [15]

On développe dans cette partie la procédure qui permet d'implémenter de manière récursive un filtre à réponse impulsionnelle infinie avec un coût algorithmique faible.

III.5.1 Opérateurs bidimensionnels de dérivation et de lissage de Deriche

Rappel : L'approche de Deriche consiste à réaliser l'opération de filtrage (convolution) sous forme de filtre à réponse impulsionnelle infini tel que :

$$d(x) = cxe^{-\alpha|x|} \qquad avec \qquad C = \frac{(1-e^{-\alpha|x|})^2}{e^{-\alpha}}$$

Le filtre utilisé est la combinaison de deux filtres unidimensionnels dans les directions x et y (l'image étant de dimension 2) . La fonction de filtrage utilisée est :

$$f(x) = b(\alpha|x|+1)e^{-\alpha|x|} \qquad avec \qquad b = \frac{(1-e^{-\alpha|x|})^2}{1+2\alpha e^{-\alpha}-e^{-2\alpha}}$$

La complexité d'implantation directe d'un produit de convolution a conduit à utiliser les filtres récursifs obtenus en utilisant la transformée en Z. L'intérêt de cette mise en œuvre récursive est le faible nombre d'opérations.

a) La dérivation :

Soit d[i] la séquence discret résultante de l'échantillonnage de $d(x) = Cxe^{-\alpha|x|}$ et D(z) sa transformée en Z tel que :

$$D(Z) = \sum_{i=-\infty}^{i=+\infty} d(i)Z^{-i}$$

$$d[i] = d^-[i] + d^+[i]$$

Avec :

$$d^-[i] = \begin{cases} Cie^{\alpha i} & si \quad i<0 \\ 0 & si \quad i=0 \end{cases}$$

$$d^+[i] = \begin{cases} Cie^{-\alpha i} & si \quad i>0 \\ 0 & si \quad i=0 \end{cases}$$

utilisant la transformée en Z du filtre correspondant est :

$$D(z) = D_-(z^{-1}) + D_+(z^{+1}) \quad avec :$$

$$D_-(z^{-1}) = \frac{Ce^{-\alpha}z^{-1}}{1 - 2e^{-\alpha}z^{-1} + e^{-2\alpha}z^{-2}}$$

$$D_+(z) = \frac{-Ce^{-\alpha}z^{+1}}{1 - 2e^{-\alpha}z^{+1} + e^{-2\alpha}z^{+2}}$$

Ou : N est la longueur de la séquence d'entrée E[i]. C la constante de normalisation.

Le signal filtré résultant est :

$$S[i] = S_1[i] + S_2[i].$$

b) Le lissage :

Soit l[i] la séquence discrète résultante de l'échantillonnage de : $l(x) = b\alpha|x| + 1e^{-\alpha|x|}$ et $L(z)$ sa transformée en Z tel que :

$$L(z) = L_-(z^{-1}) + L_+(z)$$

Avec :

$$L_-(z^{-1}) = b\frac{1 + e^{-\alpha}(\alpha - 1)z^{-1}}{1 - 2e^{-\alpha}z^{-1} + e^{-2\alpha}z^{-2}}$$

$$L_+(z) = b\frac{e^{-\alpha}(\alpha - 1)z^{+1}e^{-2\alpha}z^2}{1 - 2e^{-\alpha}z^{+1} + e^{-2\alpha}z^2}$$

Les équations aux différences auront alors pour expressions :

$$S_2[i] = bE[i] + 2e^{-\alpha}(\alpha - 1)E[i-1] + 2e^{-\alpha}S_1[i-2] - e^{-2\alpha}S_1[i-2]$$

$$S_2[i] = be^{-\alpha}(\alpha - 1)E[i+1] + be^{-2\alpha}E[i+2] + 2e^{-\alpha}S_2[i+1] - e^{-2\alpha}S_2[i+2]$$

Pour i=1..N

Le signal filtré résultant est :

$S[i] = S_1[i] + S_2[i]$

III.5.2 Le Gradient

En appliquant la même technique que précédemment, nous obtenons l'implémentation récursive de f(x). Soit f(n) les échantillons de f(x) :

$$g(n) = [c_1 sin(w|n|) + c_2 cos(w|n|)]exp(-a|n|)$$

avec $g(n) = g_-(n) + g_+(n)$ Sa transformer en Z est : $G(Z) = G_-(Z) + G_+(Z)$. Donc :

$$G_-(z) = \frac{a_0 + a_1 z^{-1}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

$$G_+(z) = \frac{a_2 + a_3 z^2}{1 + b_1 z^{+1} + b_2 z^2}$$

Avec :

$a_0 = c_2$
$a_1 = (c_1 sin(w) - c_2 cos(w))e^{-\alpha}$
$a_2 = a_1 - c_2$
$b_1 = -2e^{-\alpha} cos(w)$
$b_2 = e^{-2\alpha}$

Le calcul de Rx (gradient suivant x) et Ry (gradient suivant y) est obtenu en réalisant les convolutions [A*f*h] suivant les directions X et Y. Où A représente le signal de l'image à traiter. La discrétisation du filtre complet de Deriche nous donne les équations suivantes :

$$\frac{Rx(m,n)}{X(m,n)} = \frac{(-ce^{-\alpha|m|} sin(w|m|) * (k\alpha sin(w|n|) + w cos(w|n|)e^{-\alpha|n|})}{\alpha^2 + w^2}$$

$$\frac{Rx(m,n)}{Y(m,n)} = \frac{(-ce^{-\alpha|m|} sin(w|n|) * (k\alpha sin(w|m|) + w cos(w|m|)e^{-\alpha|m|})}{\alpha^2 + w^2}$$

L'implémentation Rx(m,n) est donnée par (avec i indice ligne et j indice colonne) :

$$y_+(i,j) = A(i,j-1) - b_1 y_+(i,j-1) - b_2 y_+(i,j-2) \qquad j = 1..n \ \ colonne$$

Avec :

$$y_-(i,j) = A(i,j+1) - b_1 y_-(i,j+1) - b2 y_+(i,j+2) \qquad i = 1..n \ \ lignes$$

la matrice de lissage est donnée par :

$$y(i,j) = a[y_+(i,j) - y_-(i,j)]$$

$$Rx_+(i,j) = a_0 y(i,j) + a_2 y(i-1,j) - b_1 R_+(i-1,j) - b_2 R_+(i-2,j)$$

$$Rx_{-}(i,j) = a_2 y(i+1,j) + a_3 y(i+2,j) - b_1 R_{-}(i+1,j) - b_2 R_{-}(i+2,j)$$

Où : j= 1...n colonnes , i = 1...m lignes.
D'ou la matrice du gradient directionnel :

$$Rx(i,j) = Rx_{+}(i,j) + Rx_{-}(i,j).$$

Les coefficients utilisés ci-dessus sont les mêmes que ceux définis précédemment.On applique la même procédure pour déterminer Ry(i,j). Les filtres présentés permettent de calculer le gradient mais ne donnent pas des points de contour.

III.5.3 Extraction des maxima locaux

L'algorithme d'extraction des maxima locaux consiste à comparer la norme du gradient NG(i,j) du point M(i,j) à celle de ses deux voisins N_{G1} du point M1 et N_{G2} du point M2 . La norme du gradient est donnée par l'expression :

$$NG(i,j) = \sqrt{Rx(i,j)^2 + Ry(i,j)^2}$$

Soit M un point de gradient G(M) et d une distance seuil (par exemple d=1). Soient M1 et M2 deux point de la droite passant par M et de vecteur directeur G(M), situés à une distance d de M. Par convention , M1 est pris dans le sens du gradient et M2 dans le sens inverse . On détermine une approximation du gradient aux points M1 et M2 par exemple par interpolation linéaire avec les points voisins.

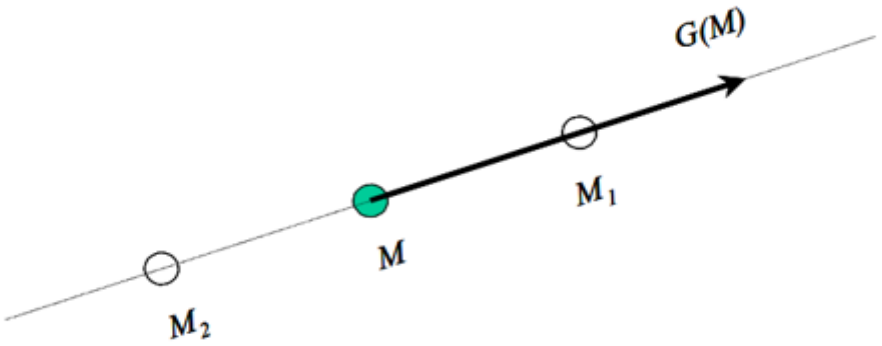


FIGURE III.5 – Extraction des maxima locaux

III.5.4 Le seuillage

Après extraction des maxima locaux, une autre opération s'applique à l'image : c'est le seuillage. Son rôle est de produire une image dont les pixels ne sont codés

que sur deux nuances de gris : le noir prend la valeur 0 et le blanc prend la valeur 255.

Dans le but de simplifier cette opération nous avons adopté une méthode statistique qui consiste à calculer la moyenne des niveaux de gris des points maxima locaux sans toutefois considérer le fond de l'image cette moyenne est donnée par :

$$S = \frac{\sum_{i=0}^{l-1} \sum_{j=0}^{c-1} Ng(i,j)}{M}$$

$$\begin{cases} N(i,j) = 255 & si \qquad N(i,j) > S \\ N(i,j) = 0 & si \qquad N(i,j) < S \end{cases}$$

III.6 Conclusion

Nous avons introduit dans ce chapitre les base de traitement d'image, ainsi que le domaine d'extraction de contours et les différentes approches(Laplacien, Gradient, Canny, Deriche.) les plus utilisées pour l'extraction de contours. Puis nous avons détaillé les étapes de filtre de Deriche. Enfin l'implémentation de filtre de Deriche.

Chapitre IV

Réalisation et expérimentation

IV.1 Introduction

L'extraction de contours d'image est une technique qui a apporté beaucoup a divers domaines d'applications notamment en médecine mais elle est souvent contrainte par le nombre de données a traiter, ce qui engendre des temps de calculs considérable. Et avec l'arrivée des architectures multi cœurs, le traitement parallèle est devenu une évidence pour palier au problème de temps.

Dans ce chapitre nous allons mettre en œuvre la parallélisation de l'algorithme d'extraction de contour d'image de Deriche dans un environnement multi cœurs, en exploitant la bibliothèque de programmation parallèle Intel TBB.

IV.2 Présentation de l'environnement de développement

Vu qu'on a choisi Intel TBB comme bibliothèque pour le développement parallèle qui est implémenté avec le langage C++, notre application a été réalisée avec C++, et Qt Creator pour le graphisme (fenêtres, buttons . . ...Etc.).

IV.2.1 Le langage C++

C++ est un langage de programmation compilé, permettant la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique. Le langage C++ n'appartient à personne et par conséquent n'importe qui peut l'utiliser sans besoin d'une autorisation ou obligation de payer pour avoir le droit d'utilisation. C++ est l'un des

langages de programmation les plus populaires, avec une grande variété de plates-formes matérielles et de systèmes d'exploitation.

Fonctionnalités introduites : On peut considérer que C++ « est du C » avec un ajout de fonctionnalités. Cependant, certains programmes syntaxiquement corrects en C++ ne le sont pas en C, à commencer bien sûr par ceux qui font usage d'identificateurs correspondant à des mots-clefs en C++.

Les fonctionnalités ajoutées sont :

- les opérateurs new et delete pour la gestion d'allocation mémoire.
- le type de données bool (booléen).
- les variables et les fonctions const.
- les fonctions inline.
- les paramètres par défaut dans les fonctions.
- les référentiels lexicaux (espaces de noms) et l'opérateur de résolution de portée « :: ».
- les classes, ainsi que tout ce qui y est lié : l'héritage, les fonctions membres, les fonctions membres virtuelles, les constructeurs et le destructeur.
- la surcharge des opérateurs.
- les templates.
- la gestion d'exceptions.
- l'identification de type pendant l'exécution (RTTI : run-time type information).
- l'inférence de type auto (C++11).
- les références de rvalue && (C++11).
- les fonctions lambda (C++11).

Avantages de C++ :

- Il est très répandu. Il fait partie des langages de programmation les plus utilisés sur la planète. On trouve donc beaucoup de documentation sur Internet et on peut facilement avoir de l'aide sur les forums.
- Il est rapide, très rapide même, ce qui en fait un langage de choix pour les applications critiques qui ont besoin de performances. C'est en particulier le cas de traitement d'image, mais aussi des outils financiers ou de certains programmes militaires qui doivent fonctionner en temps réel.

- Il est portable : un même code source peut théoriquement être transformé sans problème en exécutable sous Windows, Mac OS et Linux. Vous n'aurez pas besoin de réécrire votre programme pour d'autres plates-formes.
- Il existe de nombreuses bibliothèques pour le C++. Les bibliothèques sont des extensions pour le langage, un peu comme des plug-ins. De base, le C++ ne sait pas faire grand-chose mais, en le combinant avec de bonnes bibliothèques, on peut créer des programmes parallèles (avec intel TBB), 3D, réseaux, audio, fenêtrés, etc.
- Il est multi-paradigmes. Ce mot signifie qu'on peut programmer de différentes façons en C++. La Programmation Orientée Objet (POO) est l'une des techniques plus célèbre. C'est une technique qui permet de simplifier l'organisation du code dans nos programmes et de rendre facilement certains morceaux de codes réutilisables.

IV.2.2 Les Moyens utilisés :

Hardware : Un PC DELL avec les caractéristiques suivantes :

- Processeur intel core i5 3337U 1.8GHz :
  - . 2 cœurs physiques.
  - . 4 cœurs logiques.
  - . Cache L1 dédié 64 ko.
  - . Cache L2 dédié 256 ko.
  - . Cache L3 partagé 3 Mo.
- DDRAM 6 GO.

Software :

- Windows 7.
- Visual studio 2010.
- Qt Creator 2010.04.
- Intel TBB 4.4.

a) Les images utilisés pour l'expérimentation

Pour L'expérimentation de notre programme nous avons utilisé les images médicales montrées dans la figure IV.1 :

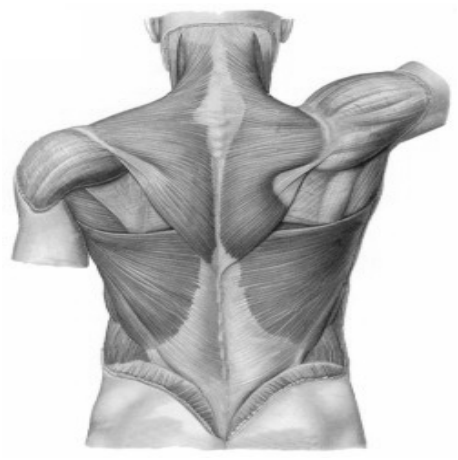


image 1 250x343

image 2 600x395

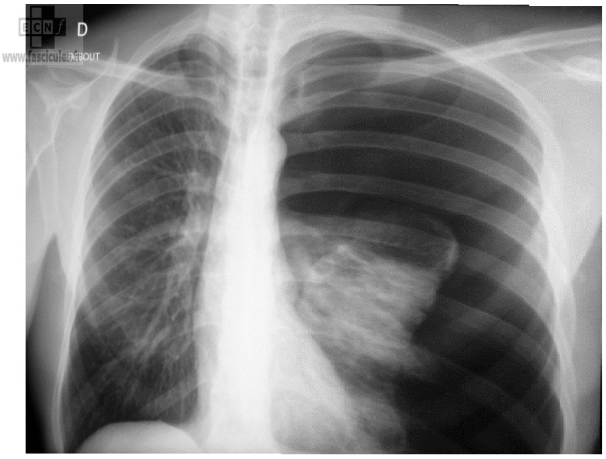


image 4 1106x668

image 3 1716x838

FIGURE IV.1 – Les images utilisées pour l'expérimentation.

IV.3 Réalisation

IV.3.1 Programme séquentielle

L'algorithme séquentiel de Deriche pour l'extraction de contours d'image est constitué essentiellement des etapes illustrées dans la figure IV.2.

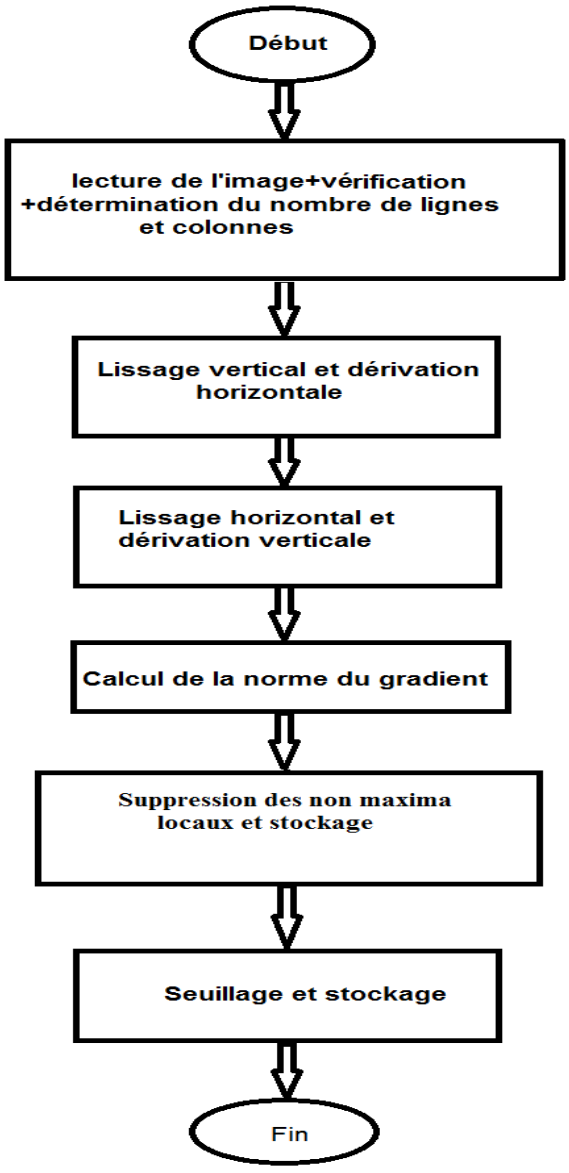


FIGURE IV.2 – Algorithme séquentiel de Deriche.

L'exécution séquentielle a donné les résultats suivants :

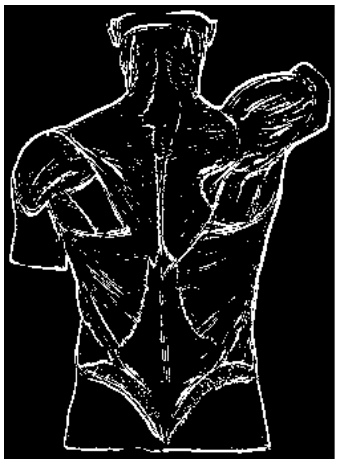


image 1

image 2

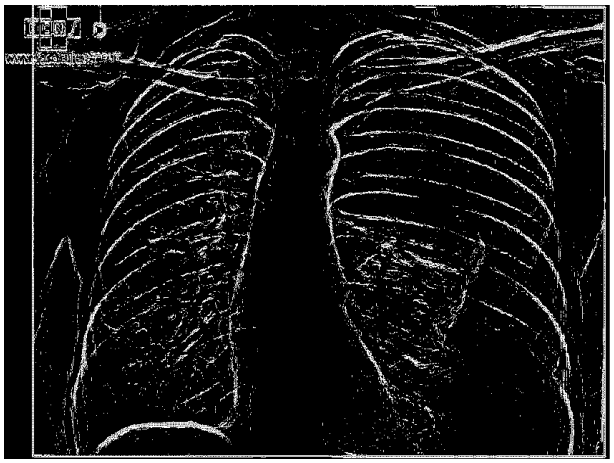


image 4

image 3

FIGURE IV.3 – Détection de contours des images avec le programme séquentiel des images.

Images	Temps d'exécution (seconds)
Image 1	0.054
Image 2	0.147
Image 3	0.406
Image 4	0.778

TABLE IV.1 – Temps d'exécution de programme séquentiel.

IV.3.2 La solution parallèle

Nous allons utiliser la décomposition par domaine. L'image sera partagée en plusieurs portions. Chaque portion sera affectée à une tâche qui calcule les contours de la portion d'image, à la fin du traitement on récupère les résultats de chaque tâche et on affiche l'image finale. Pour cela nous allons utiliser une répartition des données par bloc.

a) Répartition des données par bloc

L'image sera divisée en un nombre de bloc égal au nombre de threads disponibles sur le processeur. et les blocs seront repartis sur les threads.
La figure IV.4 illustre la repartition des données dans ce traitement parallèle.

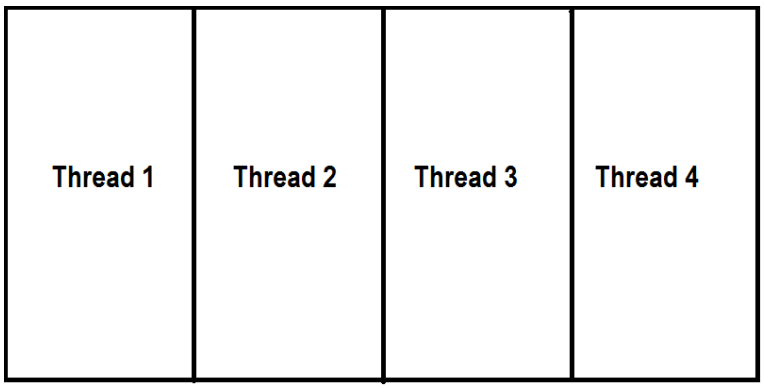


FIGURE IV.4 – répartition de données d'une image en 4 blocs affectés à 4 threads.

La figure IV.5 illustre les différentes étapes de ce traitement parallèle :

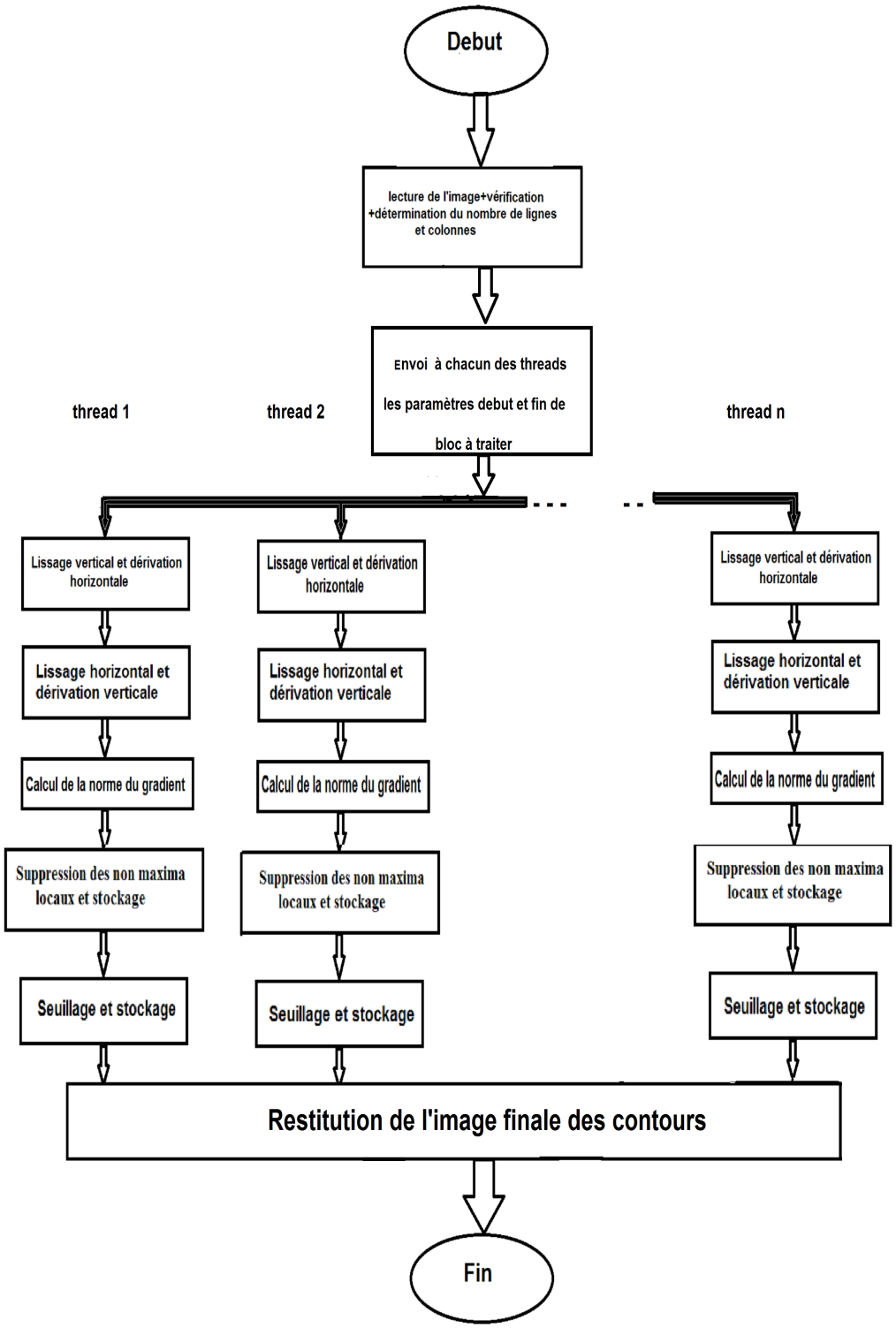


FIGURE IV.5 – Parallélisme de données sur n threads.

Pour cela, on récupère le nombre de threads sur le processeurs en faisant appel a la méthode **task_scheduler_init::default_num_threads()** qui retourne le nombre de threads sur le processeur, puis on divise l'image en un nombre de blocs égale au nombre de threads.
Pour diviser l'image on utilisera la classe **tbb::blocked_range<int>** en lui spécifiant un **grainsize** égale a la taille de chaque bloc(largeur de l'image/nombre de threads.).

La création des threads sera faite par le **task_scheduler** de Intel TBB, pour attribuer les tâches aux threads nous utiliserons la fonction **parallel_for**.

Chaque thread recevra la même tâche qui est la fonction **Deriche(dst,debut,fin)**, avec un pointeur vers l'image destination résultat et les paramètres entiers début et fin, qui représentent le début et la fin du bloc à traiter par le thread.
Le code suivant montre comment mettre en place ce parallélisme en utilisant Intel TBB :

```
1  class PDeriche{
2  public:
3          Image * src;
4          Image * dst;
5  public:
6          void operator( )( const tbb::blocked_range<int>& r) const {
7                  src->Deriche(dst,r.begin(),r.end());
8          }
9          PDeriche(Image * _src,Image * _dst):src(_src),dst(_dst){
10         }
11 };
12 void Image::ParaDericheBlocs(Image * dst,int nb_thread,float _alpha){
13         this->alpha=_alpha;
14         PDeriche pDeriche(this,dst);
15         tbb::parallel_for(tbb::blocked_range<int>(0,this->width,this->width/nb_thread),pDeriche);
16 }
```

L'exécution de ce programme a donné les résultats suivants :

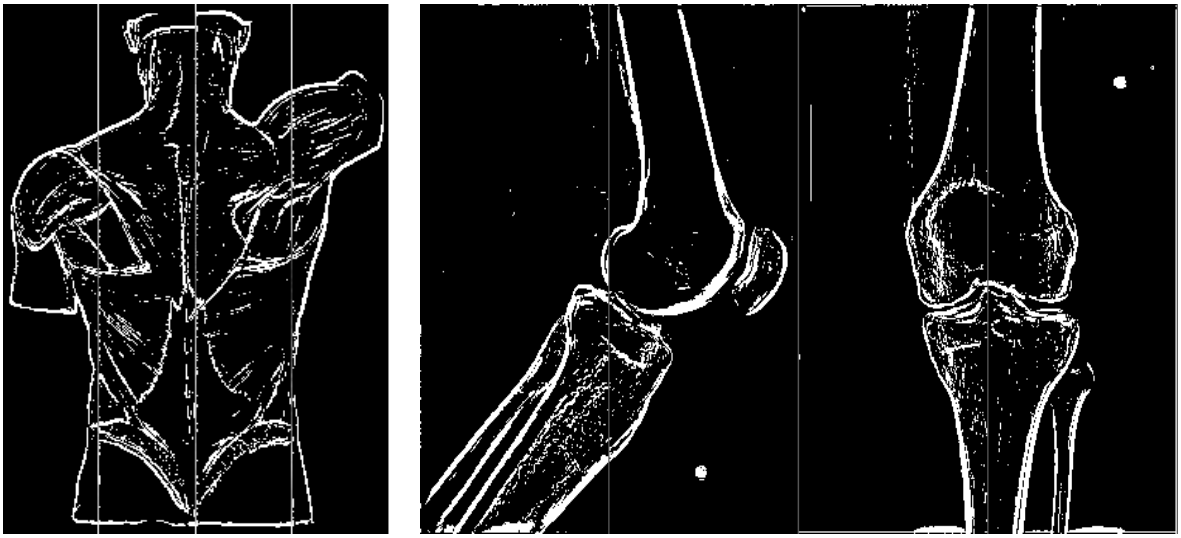


FIGURE IV.6 – Détection de contours de l'image 1 et l'image 2

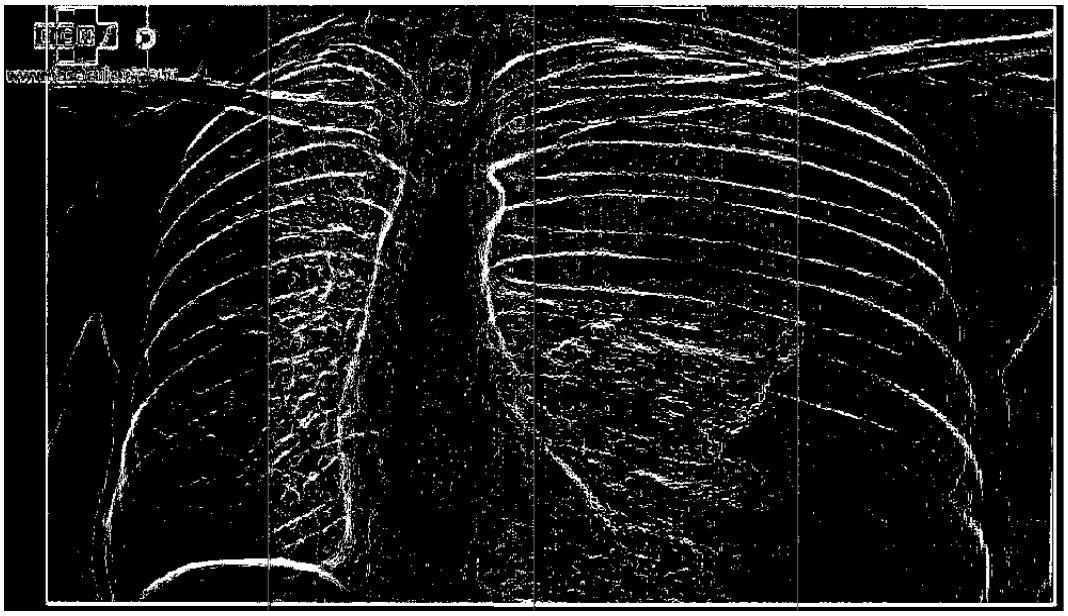


FIGURE IV.7 – Détection de contours de l'image 3

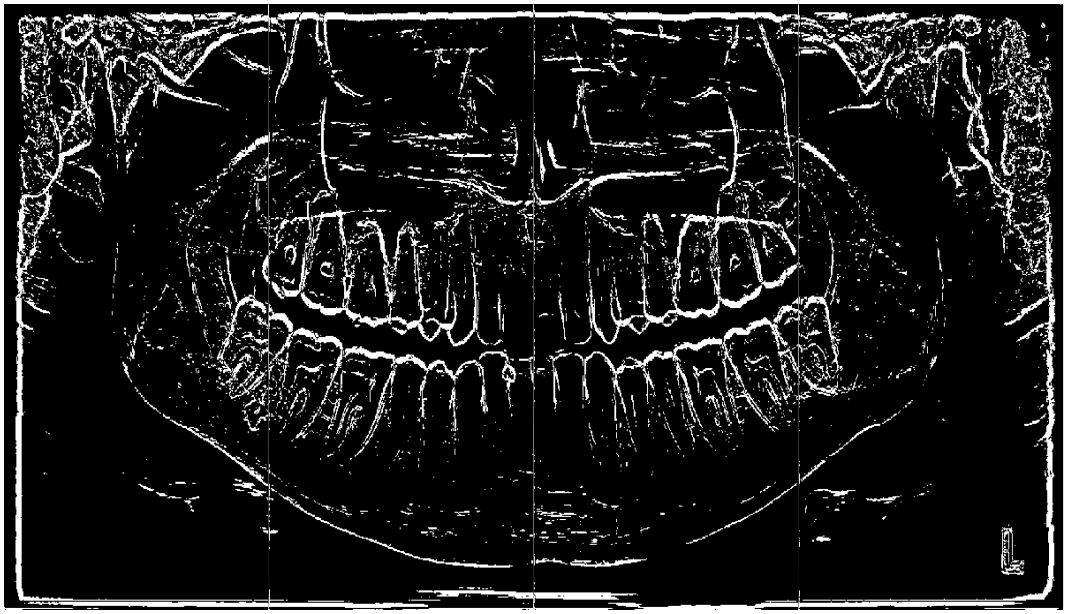


FIGURE IV.8 – Détection de contours de l'image 4

Nous constatons, sur les images précédentes, la présence de traits qui divisent ces image à l'endroit où a été effectué la segmentation par blocs de chaque image. Cela est dû au fait que le filtre de Deriche calcule la valeur d'un pixel donné en fonction de ces voisins. Pour éviter cette dégradation, il suffit de rajouter deux lignes de données aux extrémités des blocs répartis sur threads et de ne pas les prendre en considération lors de l'écriture de résultat dans l'image destination. Le traitement reste identique au précédent.
Le code suivant montre comment faire le chevauchement de données sur les bordures des blocs.

```
1  void Image::Deriche(Image * dst,int debut, int fin){
2          int n=0,m=0;
3          if(debut!=0){ n=2;debut=debut-n;}//ajout de deux colonnes au début de bloc
4          if(fin!=this->width){m=2;fin=fin+m;}//ajout de deux colonnes à fin de bloc
5          int l=fin-debut;
6          //.....
7          //extraction de contours
8          //.....
9          for (i=n;i<l-m;i++)
10                 for (j=0;j<this->height;j++)
11                         dst->ima[debut+i][j]=B[i][j];//B est le resultat de la detection de contours
12 }
```

Nous constatons aussi que la figure IV.7 fait apparaître des zones différentes, chaque zone est un blocs traité par un thread différent. Ceci est dû au fait que les blocs repartis sur chaque thread ne sont pas homogènes et par conséquent l'opération de seuillage s'est effectuée sur des niveaux différents sur chaque blocs. Pour éviter cette dégradation, il suffit de calculer un seuil unique pour toute l'image et l'appliquer sur tous les blocs. Pour cela il faut mettre en place une barrière pour synchroniser les threads.

b) La synchronisation

Pour synchroniser les threads, il faut mettre en place une barrière de synchronisation. Tous les threads effectuent leur travail et sont suspendues lorsqu'ils atteignent la barrière. Lorsque le dernier thread atteint la barrière, il calcule un seuil unique pour toute l'image, puis la barrière s'ouvre et chaque thread continue le traitement en appliquant le seuil au bloc qui lui a été attribué.

La figure IV.9 illustre les différentes étapes de traitement parallèle avec la synchronisation.

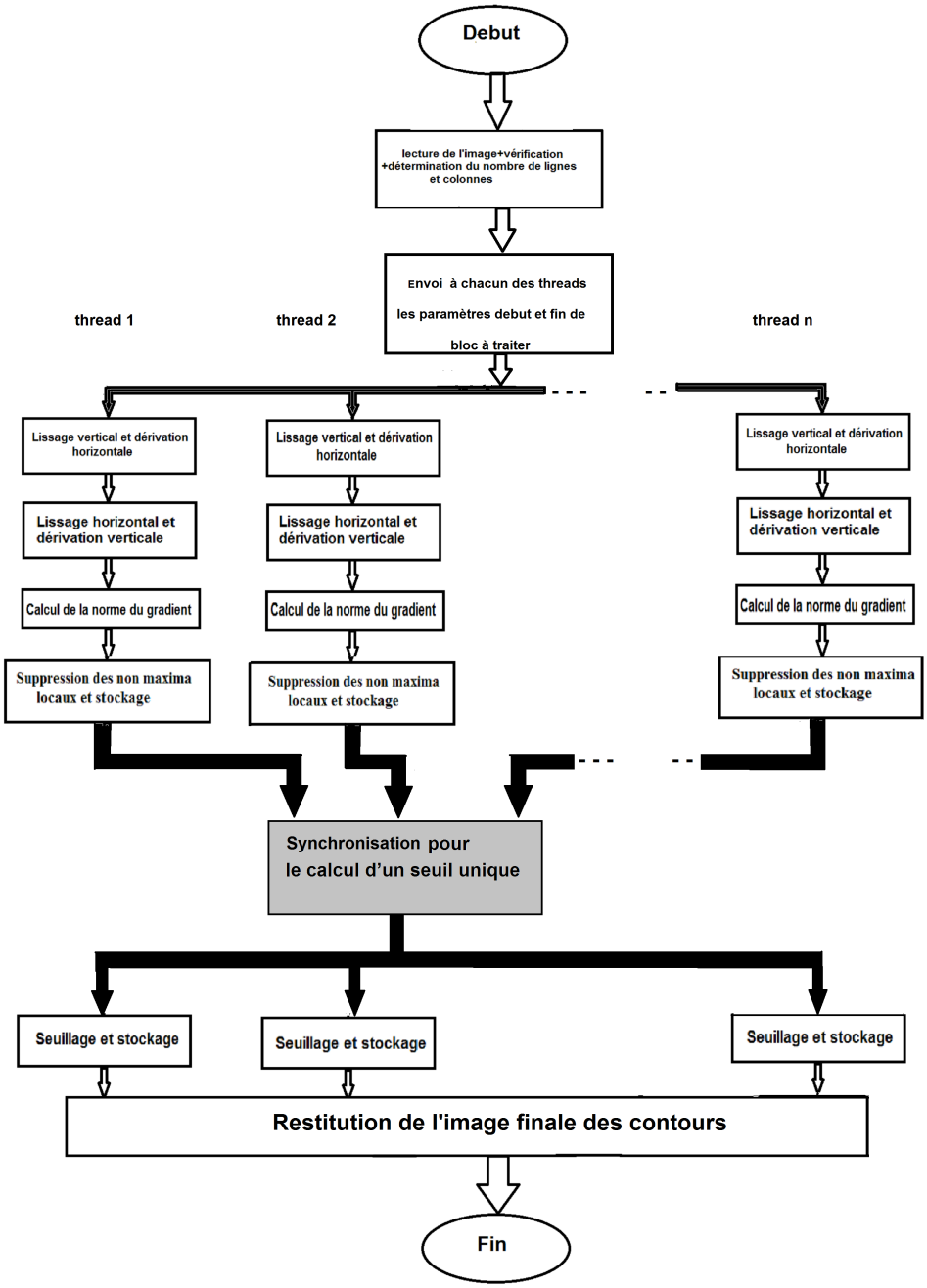


FIGURE IV.9 – Parallélisme de données sur n threads avec synchronisation.

Pour programmer cette barrière, nous avons utiliser une variable entière atomique initialisé au nombre de threads. Chaque thread qui atteint la barrière, décrémente cette variable et teste si elle est nul (dernier thread), si c'est le cas, il calcul le

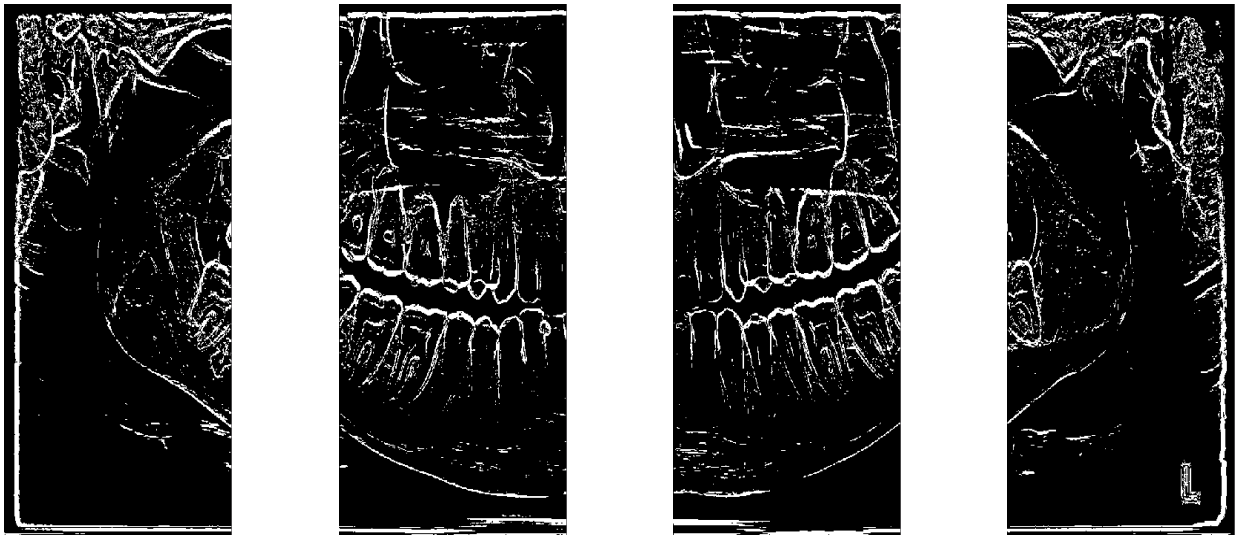
seuil et décrémente la barrière à -1. Sinon attendre jusqu'à l'ouverture de la barrière (barrière=-1).
le code suivant montre comment faire cette synchronisation et calculer le seuil.

```
1  tbb::atomic<int> bariere;
2  tbb::atomic<float> somme_ng;//somme des point maxima locaux
3  tbb::atomic<int> nb_mloco;//nombre de point maxima locaux
4  bariere=nb_thread;
5  somme_ng=0;
6  nb_mloco=0;
7  float seuille;
```

Les variable précédentes doivent être déclarées comme variable globale partagées entre les threads.

```
1  void seuillage(){
2          float local_somme_ng;//somme des point maxima locaux dans le bloc traité par chaque thread
3          float local_nb_mloco;//nombre de point maxima locaux dans le bloc traité par chaque thread
4          //.....
5          //calcul de la somme et nombre de point maxima locaux dans le bloc
6          //.....
7          somme_ng.fetch_and_store(somme_ng+local_somme_ng);
8          nb_mloco.fetch_and_add(local_nb_mloco);
9          bariere.fetch_and_decrement();
10         int i;
11         if(bariere==0)
12                 seuille=somme_ng/nb_mloco;
13                 bariere.fetch_and_decrement();
14         else
15                 while(bariere!=-1)
16                         i++;
17         //appliquer le seuil au bloc
18         //.....
19 }
```

La figure IV.10 représente les résultats de l'extraction de contours sur l'image 4 retournés par chaque thread séparément.



Block 1 Block 2 Block 3 Block 4

FIGURE IV.10 – Résultat retourné par chaque thread sur l'image 4

La figure IV.11 montre le temps d'exécution de chaque thread.

```
Thread 4408  0.38065 seconds
Thread 5404  0.381495 seconds
Thread 7328  0.38189 seconds
Thread 9988  0.383238 seconds
```

FIGURE IV.11 – Temps d'exécution de chaque thread.

Les figures IV.12, IV.13 et IV.14 représentent les contours obtenues en utilisant un chevauchement de données sur les bordures des blocs et un seuillage unique sur tous les blocs de l'image.

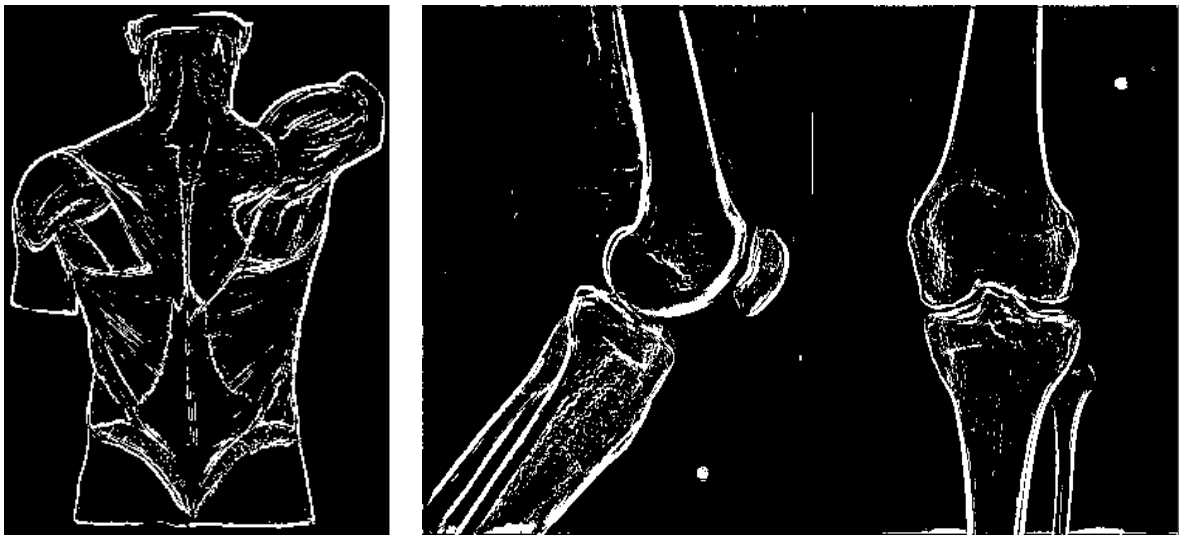


FIGURE IV.12 – Détection de contours de l'image 1 et l'image 2

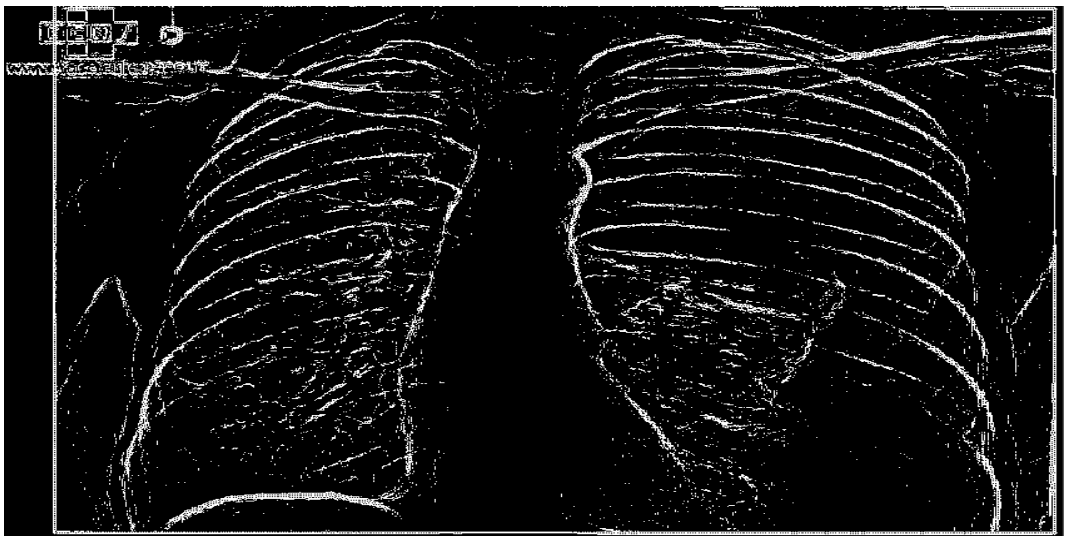


FIGURE IV.13 – Détection de contours de l'image 3

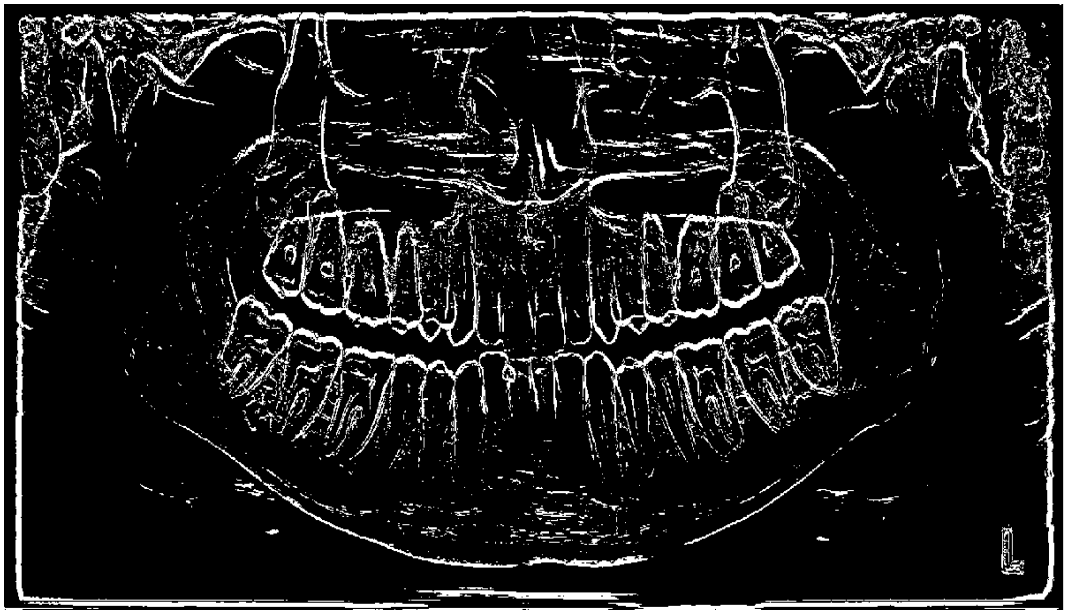


FIGURE IV.14 – Détection de contours de l'image 4

La table IV.2 représente le temps d'exécution des programmes séquentiel, et parallèle avec synchronisation et sans synchronisation.

Images	Temps d'exécution séquentiel	Temps d'exécution parallèle sans synchronisation	Temps d'exécution parallèle avec synchronisation
Image 1	0.054	0.034	0.041
Image 2	0.147	0.080	0.102
Image 3	0.406	0.182	0.214
Image 4	0.778	0.337	0.380

TABLE IV.2 – Temps d'exécution des programmes séquentiel et parallèle.

IV.4 Expérimentation

Pour récapituler les différents résultats des traitements que nous avons réalisé auparavant, nous allons représenter les différents résultats dans la table IV.3 en calculant l'accélération (Acc) et de l'efficacité (Eff) du traitement parallèle. Avec :

Accélération=temps d'exécution séquentiel/temps d'exécution parallèle
Efficacité= Accélération/ nombre de processeurs.

Image		Programme séquentiel	Programme parallèle sans synchronisation	Programme parallèle avec synchronisation
Image1 250x343	Temps	0.054	0.034	0.041
	Acc	1	1.58	1.31
	Eff	1	0.39	0.32
Image2 600x395	Temps	0.147	0.080	0.102
	Acc	1	1.83	1.44
	Eff	1	0.45	0.36
Image3 1106x668	Temps	0.406	0.182	0.214
	Acc	1	2.23	1.89
	Eff	1	0.55	0.47
Image4 1716x838	Temps	0.778	0.337	0.380
	Acc	1	2.30	2.04
	Eff	1	0.57	0.51

TABLE IV.3 – Résultats du traitement parallèle

A partir de ces résultats, nous remarquons que le traitement parallèle est toujours plus performant qu'un traitement séquentiel. L'accélération qui exprime le rapport du temps de traitement séquentiel au temps de traitement parallèle est toujours supérieure à 1. Nous remarquons aussi que la mis en place de la synchronisation ralenti le traitement parallèle.

Nous remarquons également que les temps de traitement avec un parallélisme sans synchronisation sont plus performants que ceux d'un parallélisme avec synchronisation. Mais dans le cas d'images non homogènes, la synchronisation reste incontournable afin d'assurer des meilleurs résultats.

L'accélération et l'efficacité augmentent avec l'augmentation de la taille de l'image utilisé. Pour les petites images l'accélération est moins convaincante par rapport aux grandes images, car dans ces petites images le coût de la mise en place de parallélisme est important par rapport au temps du traitement effectif sur les données.

IV.5 Présentation de l'application

L'application se présente comme une fenêtre GUI, permettant à l'utilisateur de sélectionner une image à partir de son disque dur, ensuite appliquer le filtre de détection de contours, le traitement peut se faire d'une manière séquentielle ou parallèle, en suite enregistrer le résultat sur son disque dur.

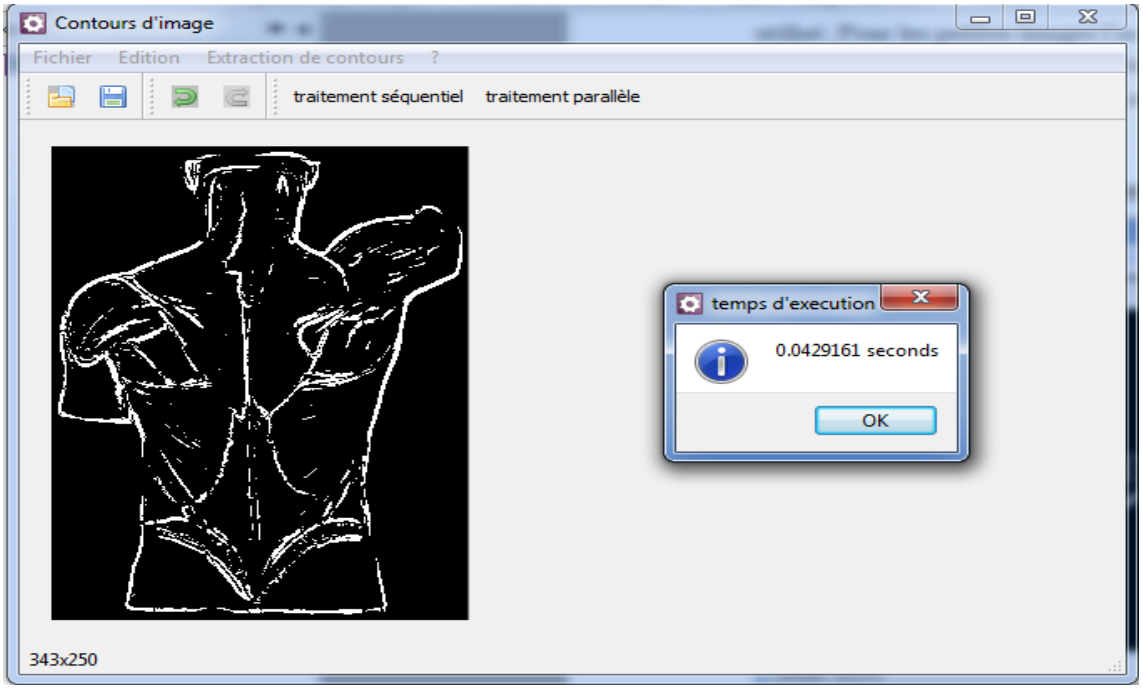


FIGURE IV.15 – Présentation de l'application

IV.6 Conclusion

Dans ce chapitre nous avons vu la mise en œuvre de la parallélisation de l'algorithme de Deriche. Nous avons présenté notre environnement de développement, les moyens utilisés. nous avons proposé deux solution pour le traitement parallèle. la première est un parallélisme de données sans synchronisation, et la deuxième est un parallélisme de données avec synchronisation.

Nous avons aussi vu que les deux solutions sont plus performante en terme de temps par rapport au traitement séquentiel. Le traitement parallèle sans synchronisation donne de bons résultats en terme d'accélération du temps d'exécution, mais pour certaines images qui comportent des zones hétérogènes cette solution ne donne pas une meilleure qualité de l'image des contours. Pour avoir de meilleurs résultats, nous avons proposé de synchroniser les threads pour calculer un seuil unique. Par contre cette solution est moins performante par rapport à la première. Le traitement parallèle sans synchronisation nous a donné une accélération qui varie entre 1.58 et 2.30, et le traitement parallèle avec synchronisation nous a donné une accélération qui varie entre 1.31 et 2.04.

Nous avons aussi constaté que le traitement parallèle est plus efficace sur des image de grande taille par rapport aux images de petite taille.

Conclusion générale

Les processeurs ont connu une évolution sans précédent avec l'arrivée des architectures multi cœurs. Ce type d'architecture permet l'augmentation des performances des processeurs tout en apportant une solution aux difficultés de dissipation thermique. Pour tirer profit de cette évolution, les applications doivent exploiter le parallélisme offert par les architectures multi cœurs.

Au cours de la réalisation de ce mémoire, nous avons exploité le parallélisme sur un processeur multi cœurs en utilisant la bibliothèque **Intel TBB**, qui facilite la programmation parallèle grâce aux différents composants qu'elle fournit.

D'abord, nous nous sommes penchés sur l'étude de l'architecture des processeurs multi cœurs et les notions de base sur le parallélisme dans les systèmes multi cœurs. Nous avons approfondis nos connaissances dans la programmation parallèle en utilisant la bibliothèque **Intel TBB** et exploiter les facilitations qu'elle offre pour la réalisation. Ce travail nous a permis de découvrir le domaine de l'imagerie en particulier les traitements pour l'extraction des contours d'image qui est un domaine d'actualité et très intéressant.

Nous avons réussi la réalisation d'une application qui exploite le parallélisme offert par les processeurs multi cœurs pour l'extraction des contours d'image. Nous avons constaté que le traitement parallèle a permis d'obtenir un gain sur le temps de traitement qui varie entre 24% à 51% par rapport au temps d'un traitement séquentiel sur un processeur à 4 cœurs. Les résultats de notre expérimentation, nous permettent de déduire que l'efficacité du traitement parallèle est meilleure sur les images de grande taille.

Pour conclure, la réalisation de ce travail nous a donné l'occasion d'acquérir des

connaissances intéressantes concernant le traitement parallèle. Nous avons eu aussi l'occasion d'apprendre comment paralléliser un code séquentiel et les avantages qu'offre le traitement parallèle par rapport à un traitement séquentiel, telles que l'accélération du temps d'exécution des programmes et l'exploitation optimale des ressources matérielles disponibles.

Bibliographie

[1] Tarik Saidani. *Optimisation multi-niveau d'une application de traitement d'images sur machines paralléles*. PhD thesis, Université Paris-Sud, 2012.

[2] Blaise Barney. Introduction to parallel computing. https ://computing.llnl.gov/tutorials/parallel_comp[en ligne].

[3] Khaled BELKACEM. Parallélisme dans les applications desktop, 2011. Mémoire de fin d'études, UMMTO.

[4] wikipedia. https ://wikipedia.org.

[5] Généralités sur le parallélisme. https ://ciment.ujf-grenoble.fr/mirage/parallelisme/generalites-sur-le-parallelisme[en ligne].

[6] Sylvain Geneves. *Améliorations de performances des systèmes multi-cœur : environnement d'exécution événementiel efficace et étude comparative de modèles de programmation*. PhD thesis, Université de Grenoble, 2012.

[7] Vajda András. *Programming Many-Core Chips*. Springer, 2011.

[8] Jahagirdar Sanjeev, George Varghese, Sodhi Inder, and Wells Ryan. Power management of the third generation intel core micro architecture formerly codenamed ivy bridge. http ://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf.

[9] Dan Negrut. Elements of processor architecture, 2013. http ://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf.

[10] James Reinders. *Intel Threading Building Blocks,Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[11] Diane Lingrand. Cours de le traitement d'images, 2004.

[12] Lyes IZABACHENE. Accès distant et traitement parallèle sur des données distribuées, 2008. Mémoire de fin d'études, UMMTO.

[13] Pierre-Arnaud CHASTANET. Filtre de shen ou de deriche pour détecter un contour ? http ://www.tsi.telecom-paristech.fr/pages/enseignement/ressources/beti/Shen_ou_Deriche/mti.html [en ligne].

[14] Mazières Bertrand and Pontini Olivier. Le filtre de deriche est-il isotrope ? http ://www.tsi.telecom-paristech.fr/pages/enseignement/ressources/beti/deriche_isotrope/deriche.html [en ligne].

[15] A Taguemount, L Djema, and F.O Boumghar. Traitement parallèle sous mpi-2, pour l'accélération de l'algorithme d'extraction de contours d'images., Mars 2005,Tunisie.