R ÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

UNIVERSITÉ DE TIZI-OUZOU
DEPARTEMENT D'INFORMATIQUE

Thème :

Traitement parallèle sous environnement multicœur.

Projet mémoire de fin du cycle mastère.

Réalisé & soutenu publiquement par :

M^{lle} CHEKROUN Djurdjura &

M^{r} CHEMLOUL Nassim.

Le : ….. / ….. / …..…..

En vue de l'obtention du

« **Grade de Master 2 en informatique, option ''Systèmes Informatiques''** ».

Sous la direction de :

M^{r} L, DJEMA.

Devant le jury composé de :

- Président : .....
- Encadreur :.....
- Examinateur :.....
- Examinateur :.....

2016/2017

R ÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

UNIVERSITÉ DE TIZI-OUZOU
DEPARTEMENT D'INFORMATIQUE

Thème :

Traitement parallèle sous environnement multicœur.

Projet mémoire de fin du cycle mastère.

Réalisé & soutenu publiquement par :

M[lle] CHEKROUN Djurdjura &

M[r] CHEMLOUL Nassim.

Le : ….. / ….. / …..…..

En vue de l'obtention du

« **Grade de Master 2 en informatique, option ''Systèmes Informatiques''** ».

Sous la direction de :

M[r] L, DJEMA.

Devant le jury composé de :

- **Président :** .....
- **Encadreur :**.....
- **Examinateur :**.....
- **Examinateur :**.....

2016/2017

À la mémoire de notre regretté

professeur DEMRI, paix à son âme.

Remerciements

De prime abord, nous tenons à remercier Mr L.DJEMA, notre promoteur, de nous avoir encadrés tout au long de ce projet.

Que les membres du jury trouvent ici nos remerciements les plus vifs pour avoir accepté d'honorer par leurs jugements ce modeste travail.

À tous les enseignants qui ont assuré notre formation durant notre parcours universitaire, pour l'ensemble des connaissances qu'ils nous ont transmis ; nous souhaitons qu'en ces quelques mots ils trouvent l'expression de notre infinie reconnaissance.

Nous souhaitons également exprimer notre profonde gratitude à tous ceux et celles qui de près ou de loin ont apporté leur précieuse aide dans la réalisation du présent travail.

TABLE DES MATIERES

TABLE DES MATIERES

TABLE DES MATIERES

TABLE DES MATIERES

LISTE DES FIGURES

Chapitre I :

Chapitre II :

LISTE DES FIGURES

Chapitre III :

LISTE DES FIGURES

Chapitre IV :

LISTE DES TABLEAUX

Chapitre I :

Chapitre III :

Chapitre IV :

INTRODUCTION GÉNÉRALE

Introduction générale :

Les besoins croissants en puissance de calcul et la nécessité de résoudre des problèmes de grande taille ont été au cœur des motivations des concepteurs de microprocesseurs. Ces derniers ont d'abord tenté d'augmenter les fréquences d'horloge des microprocesseurs afin d'accroître leurs performances, mais cette solution entraina beaucoup de problèmes insurmontables et fût alors abandonnée. L'autre solution retenue a été donc, de multiplier le nombre de cœurs sur les microprocesseurs.

Or, les programmes codés jusqu'alors ne bénéficient pas automatiquement de la puissance des microprocesseurs multicoeurs. Ce qui amène les programmeurs à se tourner vers une nouvelle approche de programmation appelée « parallèle » qui sied parfaitement aux multicœurs.

L'extraction de contours d'images est une technique qui requiert une grande puissance de calcul et qui se prête parfaitement au traitement parallèle. Son but est de réduire la quantité de données en éliminant les informations jugées être les moins pertinentes, tout en préservant les propriétés structurelles importantes de l'image.

Notre travail consiste à implémenter, sous Qt, l'algorithme de R.Deriche servant à l'extraction de contours d'images, dans une application desktop, en exploitant le parallélisme sur des microprocesseurs multicœurs.

Notre mémoire est organisé en deux parties, la première partie intitulée « revue bibliographique » est composée du chapitre I exposant les architectures multicoeurs et le parallélisme des programmes. Le chapitre II, étant quant à lui, consacré à la technique d'extraction de contours selon l'algorithme de R.Deriche.

La deuxième partie nommée « de la méthodologie à l'expérimentation » englobe le chapitre III dans lequel sont présentés les outils de développement, le langage C++ et le « *framework* » Qt, et enfin le chapitre IV consacré à la réalisation de notre application puis discussion des résultats de son expérimentation.

PREMIÈRE PARTIE :
REVUE BIBLIOGRAPHIQUE

Chapitre I : Architecture multicœur & parallélisme.

Dans ce chapitre, nous essayerons de comprendre les raisons qui ont dévié l'évolution des microprocesseurs de la course aux gigahertz vers les architectures multicoeurs, puis d'appréhender la programmation parallèle qui bénéficie du passage à plusieurs cœurs.

I.A) Architecture de base d'un ordinateur :

L'architecture d'un ordinateur représente l'organisation de ses différentes unités et leurs interconnexions [01].

L'architecture de base d'un ordinateur est celle de John VON NEUMANN, élaborée en 1945. Depuis, la quasi-majorité des ordinateurs s'y conforme [02-03].

I.A.1) Architecture de VON NEUMANN :

Le terme « Architecture de VON NEUMANN » est néanmoins discuté vis-à-vis des collaborateurs de John VON NEUMANN, notamment John William MAUCHLY et Prosper ECKERT, il est donc maintenant plutôt proscrit [03-04].

L'architecture de VON NEUMANN est composée des éléments suivants :

- Une unité centrale ;
- Une mémoire principale ;
- Une interface d'entrées/sorties [01].

Ces 3 éléments sont reliés par un moyen de communication, appelé « bus » [01].

La figure I.01 représente l'architecture de VON NEUMANN :

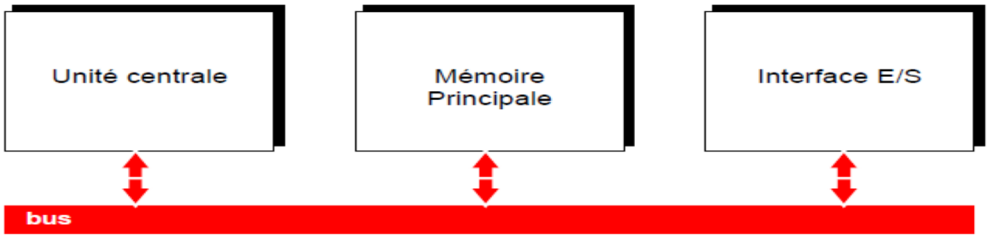

Figure I.01 : Architecture de VON NEUMANN [01].

I.A.1.a) L'unité centrale :

L'unité centrale est un microprocesseur qui exécute séquentiellement les instructions du programme [01].

I.A.1.b) La mémoire principale :

Elle contient à la fois les données et les instructions du programme. Ces dernières indiquent au microprocesseur quels sont les calculs à faire sur les données [01,03].

La figure I.02 représente la mémoire principale, dont l'espace est partagé entre le programme & les données :

Figure I.02 : Mémoire principale partagée par programme & données.

I.A.1.c) L'interface d'entrées/sorties :

Elle permet d'assurer la communication entre le microprocesseur et les périphériques, comme : le clavier (en entrée) ou l'imprimante (en sortie) [01,03].

Ces périphériques permettent à leur tour une interaction entre l'ordinateur et l'utilisateur. Ceux d'entrées permettant à l'utilisateur d'envoyer des données vers l'ordinateur, par opposition à ceux de sorties lui permettant d'extraire des données de l'ordinateur [04].

La figure I.03 représente l'interaction entre l'utilisateur, les périphériques & l'interface d'entrées/sorties :

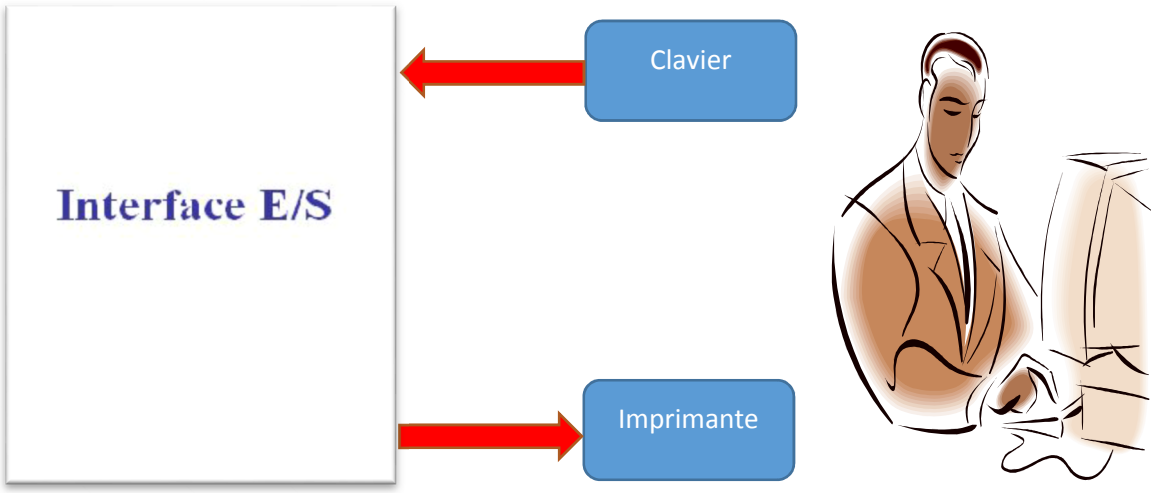

Figure I.03 : Interaction utilisateur, périphériques et interface d'E/S.

I.B) Exécution du programme :

Comme un programme n'est autre qu'une succession d'instructions, alors pour que le microprocesseur l'exécute, il n'aura qu'à exécuter séquentiellement ses instructions.

Pour exécuter une instruction, le microprocesseur alterne entre « Accès mémoire » et « Traitement », en effectuant successivement les opérations suivantes :

- Lire l'instruction en mémoire principale « Accès mémoire » ;
- Décoder l'instruction « Traitement »;
- Lire les données demandées par l'instruction en mémoire principale « Accès mémoire » ;
- Effectuer le traitement demandé par l'instruction sur les données lues « Traitement » ;
- Écriture des données résultantes du traitement en mémoire principale « Accès mémoire ».

I.C) Réduction du temps d'exécution d'une instruction :

Comme vu précédemment, le temps d'exécution d'une instruction égale la somme du temps des accès mémoire et du temps des traitements [05].

Temps d'exécution d'une instruction = Temps accès mémoire + Temps traitements (1)

Pour réduire le temps d'exécution d'une instruction, il faut alors réduire le temps des accès mémoire et/ou le temps des traitements.

I.C.1) Réduction du temps des accès mémoire :

Le temps des accès mémoire peut être réduit par l'utilisation d'un cache [06].

Le cache est une mémoire s'intercalant entre le microprocesseur et la mémoire principale, comme le montre la figure I.04 :

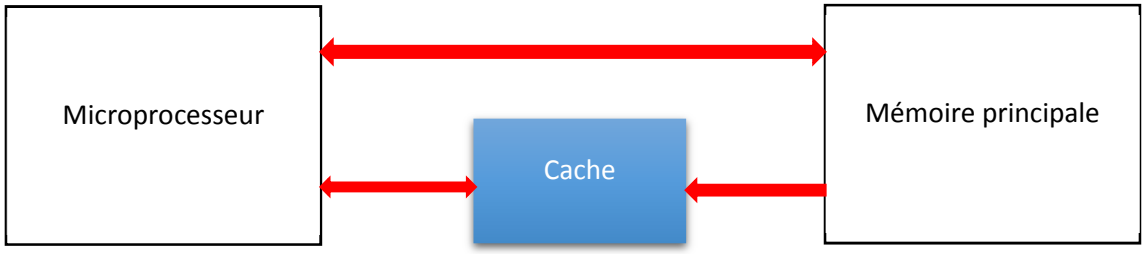

Figure I.04 : Le cache.

Le cache est une image restreinte de la mémoire principale, contenant les informations[1] les plus susceptibles d'être utilisées par le microprocesseur dans un délai très proche [07].

Le cache est plus rapide que la mémoire principale. Et cela pour 2 raisons :

1- Sa proximité du microprocesseur ;
2- Sa nature statique : n'ayant pas besoin d'un rafraichissement[2] périodique, de sorte qu'il soit plus disponible. Contrairement à la mémoire principale souvent de nature dynamique.

Principe de fonctionnement du cache :

Lorsque le microprocesseur a besoin d'une information, il la recherche d'abord dans le cache :

- Si celle-ci s'y trouve alors elle est lu rapidement « Succès de cache » ;
- Si celle-ci ne s'y trouve pas « Défaut de cache », alors une lecture de la mémoire principale est nécessaire. Ensuite, l'information recherchée ainsi que des informations adjacentes sont, dès lors, transférées de la mémoire principale au cache. De sorte que les futures références à la mémoire trouvent rapidement les informations nécessaires dans le cache, selon « Le principe de localité spatiale »[3] [06].

[1] Les instructions & les données.
[2] De recevoir du courant électrique pour ne pas perdre les informations qu'on lui a confiées.
[3] Les informations voisines à l'information recherchée seront probablement bientôt utilisées.

Selon le principe de fonctionnement du cache tout juste énoncé, lorsqu'un «Défaut de cache » se produit, une perte de temps est enregistrée, du fait que le microprocesseur consulte le cache pour rien. Ce qui nous amène au calcul du « Taux de succès », lequel se doit être supérieur à 0.9 pour que le cache tienne toutes ses promesses [06-07].

Le « Taux de succès » se calcul comme suit :

$$Taux\_de\_succès = Nombre\_de\_succès / Nombre\_accès > 0.9 \quad (2)$$

Niveaux de cache :

On distingue 3 niveaux de cache, du niveau 1 (L1 : pour *Level 1*) au niveau 3 (L3).

Plus on monte dans ces niveaux :

- Plus la distance les séparant du microprocesseur s'étend => Plus lentes sont leurs vitesses ;
- Plus leurs coûts diminuent => plus leurs capacités augmentent.

La figure I.05 appui ceci :

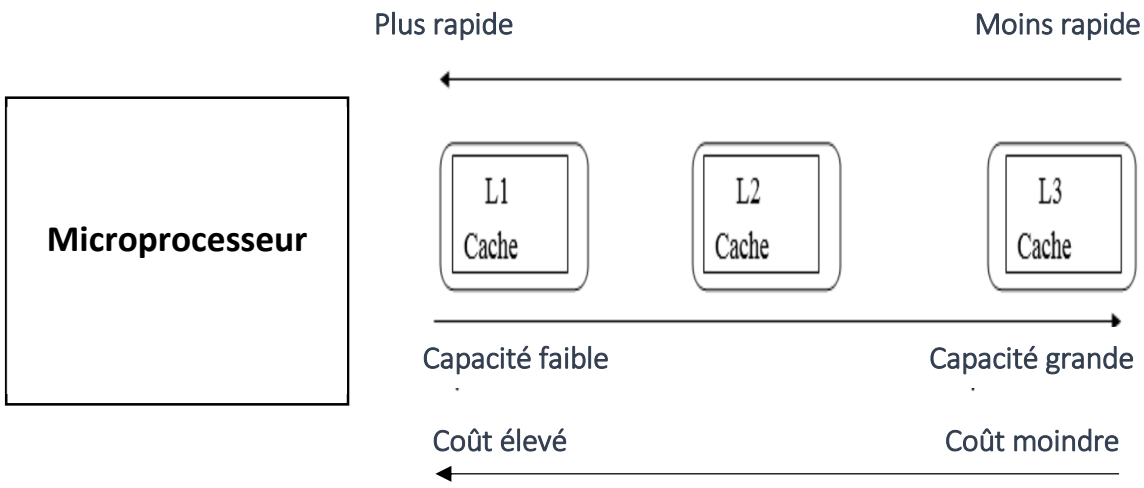

Figure I.05 : Les niveaux de caches.

Concernant le cache L1, le cache de données est souvent séparé du cache d'instructions.

I.C.2) Réduction du temps des traitements :

Le temps des traitements dans le microprocesseur peut être réduit en augmentant la fréquence d'horloge de celui-ci.

I.C.2.a) Augmentation de la fréquence d'horloge :

Toutes les activités du microprocesseur sont cadencées par une horloge, comme pour la plupart des circuits électroniques complexes, on dit d'eux qu'ils sont synchrones[1] [07-09].

L'horloge est un cristal de quartz qui, soumit à un courant électrique, envoie des impulsions. À chaque impulsion, une tâche élémentaire[2] est réalisée par le microprocesseur [05].

La fréquence d'horloge, donnée en Hertz (Hz), notée ***f***, est le nombre d'impulsions de l'horloge pendant 1seconde. Elle est calculée comme suit :

$$f = 1/T \quad (3), \text{ avec } \boldsymbol{T} : \text{le cycle d'horloge.}$$

Le cycle d'horloge ***T*** est le temps écoulé entre 2 impulsions émises par l'horloge. La durée de ce cycle est choisie en fonction de la durée maximale possible de chacune des tâches élémentaires réalisée par le microprocesseur, afin d'assurer qu'à la prochaine impulsion toutes sont terminées et que les résultats produits sont stables [08].

L'équation (3) montre qu'à architecture équivalente, si la fréquence ***f*** augmente, alors ***T*** diminue. Ce qui signifie que la tâche élémentaire est exécutée en moins de temps.

Exemple :

Voici une comparaison entre 2 microprocesseurs d'architectures équivalentes :

1- Cas 1 : Le microprocesseur 8086 d'Intel.
f_1 = 05 MHz => $T_1 = 1 / (05*10^6) = 2.0*10^{-7}$ secondes => 1 tâche élémentaire en 200 ns.
2- Cas 2 : Le microprocesseur Pentium 100 d'Intel.
f_2 = 66 MHz => $T_2 = 1 / (66*10^6) = 1.5*10^{-8}$ secondes => 1 tâche élémentaire en 015 ns.
En somme, ($f_2 > f_1$) ⇔ ($T_2 < T_1$).

Pour augmenter la fréquence d'un microprocesseur, les fabricants gravent des composants de plus en plus petits, de telle sorte que le courant les traverse aussi vite que possible. Or ces composants chauffent, et leur taille de plus en plus petite concentre d'autant plus la chaleur sur le microprocesseur [10].

Cette surchauffe entraine des dysfonctionnements majeurs dans le microprocesseur (Erreurs de calcul, redémarrages intempestifs …etc.) allant jusqu'à sa détérioration [11].

Sachant que le microprocesseur est doté d'un système de refroidissement permettant de dissiper la chaleur produite. Une première approche développée par les fabricants était d'améliorer ces systèmes afin qu'ils puissent dissiper plus de chaleur.

---

[1] Par opposition aux circuits n'usant pas d'horloge, appelés asynchrone.
[2] Un traitement (comme décoder une instruction) dans le microprocesseur est constitué d'une multitude de tâches élémentaires.

Systèmes de refroidissement :

- **Refroidissement à air :** Consiste en un dispositif appelé ventirad « ventilateur + radiateur » apposé au-dessus du microprocesseur.
  Inconvénient : si l'on veuille que le ventirad dissipe plus de chaleur, il faudrait augmenter la vitesse de rotation de son ventilateur. Ce qui implique une nuisance sonore non négligeable ;
- **Refroidissement à eau :** est un dispositif faisant circuler de l'eau, bien meilleure conductrice thermique que l'air, dans un « *waterblock* » (Bloc d'eau). Ce « *waterblock* », situé sur le microprocesseur, permet un transfert thermique entre l'eau et le microprocesseur. Le silence de fonctionnement est un autre atout du refroidissement à eau.
  Inconvénient : Cette méthode est dangereuse à cause de la cohabitation entre l'eau et l'électricité [11].

En somme, ces systèmes cumulent beaucoup trop d'inconvénients, auquel on ajoute le surplus de consommation énergétique causé, surtout pour les dispositifs mobiles ou l'autonomie de ces derniers étant un gage de qualité à préserver coute que coute.

Pour toutes ces raisons, cette approche fut abandonnée, et par-delà la course aux gigahertz le fut elle aussi. Par la suite, les fabricants de microprocesseurs se sont tournés vers un tout autre champ de recherche « Les architectures multiprocesseurs ou multicœurs ».

Le fabricant de microprocesseurs Intel a abandonné l'objectif avancé en 2002 d'atteindre les 10 GHz avant fin 2005. Ses microprocesseurs devraient rester en dessous des 4 GHz. Le dernier modèle produit par la firme à être surcadencé à 3.8 GHz fut le Pentium 4C.

Un porte-parole d'Intel explique : « Nous atteignons une limite en terme de montée en fréquences, synonyme d'augmentation de chaleur. ». Et enchaine : « Nous pourrions basculer vers de nouveaux systèmes de refroidissement, mais ils consomment beaucoup d'électricité. La meilleure option est donc de multiplier les cœurs sur un même microprocesseur. » [12].

Ce revirement s'explique par le fait qu'un microprocesseur :

- Auquel double-t-on de fréquence, doublera de performance et quadruplera la quantité de chaleur générée ;
- Auquel double-t-on de cœur, doublera de performance et par contre ne fera que doubler la quantité de chaleur générée [13].

I.D) Architecture multicœurs :

Les microprocesseurs sont gravés sur des plaques de silicium appelées « *wafers* » [07].

Cœur : est la partie centrale du microprocesseur. Il possède tout le nécessaire pour l'exécution séquentielle d'un programme (unité de calcul, cache …etc.) [07,13].

Microprocesseur multicœurs : possède plusieurs cœurs physiques[1] gravés sur un même « *wafer* ». Cela est devenu possible grâce à l'augmentation de la finesse de gravure dans les générations successives des microprocesseurs [07].

Le tableau I.1 montre l'évolution de la finesse de gravure des microprocesseurs d'Intel :

Tableau I.1 : L'évolution de la finesse de gravure des microprocesseurs d'Intel [10].

Année de commercialisation	Nom	Finesse de gravure (µm)	Nombre de cœurs
1980	8088	3.0	1
1993	Pentium	0.8	1
2000	Pentium 4C	0.18	1
2005	**Core 2 Duo**	**0.065**	**2**

Comme énoncé précédemment, chacun des cœurs du microprocesseur est capable de traiter des instructions individuellement [14]. Donc, sur un seul microprocesseur multicœur, plusieurs instructions peuvent être traitées simultanément. C'est ainsi que l'augmentation des performances des architectures multicœurs est enregistrée.

Le premier microprocesseur multicœur à être commercialisé fut le POWER 4 d'IBM en 2001, quatre ans plus tard, Intel et AMD ripostèrent [13].

Comparaison entre l'architecture multiprocesseur et multicœur :

Le tableau I.2 résume les principales différences entre ces 2 architectures :

Tableau I.2 : Les différences entre l'architecture multiprocesseur et multicœur.

Architecture multiprocesseur	Architecture multicœur
Plusieurs microprocesseurs physiquement indépendants.	Un seul microprocesseur contenant plusieurs cœurs gravés sur un seul « *wafer* ». Ce qui réduit considérablement les coûts de production, permettant aux fabricants d'effectuer d'autres améliorations tout en gardant des prix abordables.
La séparation de l'information se fait par le bus système FSB «*Front Side Bus* ».	La séparation de l'information se fait par le bus microprocesseur CMP « *Chip Level Multiprocessing* ». Ce qui réduit la distance entre les cœurs, augmentant ainsi la vitesse des échanges.

[1] En opposition aux cœurs logiques émulés dans la technique du multithreading.

À présent que nous avions énoncé les concepts fondamentaux du sujet, nous passons, de suite, à l'architecture multicœur en détaillant les points suivants :

1) Les schémas d'interconnexions des cœurs du microprocesseur ;
2) Les multicœurs homogènes/hétérogènes ;
3) L'organisation du cache dans les multicœurs ;
4) Les accès à la mémoire principale par les multicœurs ;
5) Programmation pour les multicœurs.

I.D.1) Les schémas d'interconnexions des cœurs du microprocesseur :

Les cœurs d'un microprocesseur sont reliés par des bus pour pouvoir communiquer entre eux. Ces bus sont organisés selon un schéma d'interconnexion.

Il existe actuellement plusieurs schémas d'interconnexion, dont les plus répandus, sont :

- En bus commun « *Bus* » ;
- En grille « *Mech* »;
- En anneaux « *Ring* » ;
- En barre transversale « *Crossbar* » [15].

La figure I.06 donne un aperçu sur ces schémas d'interconnexion :

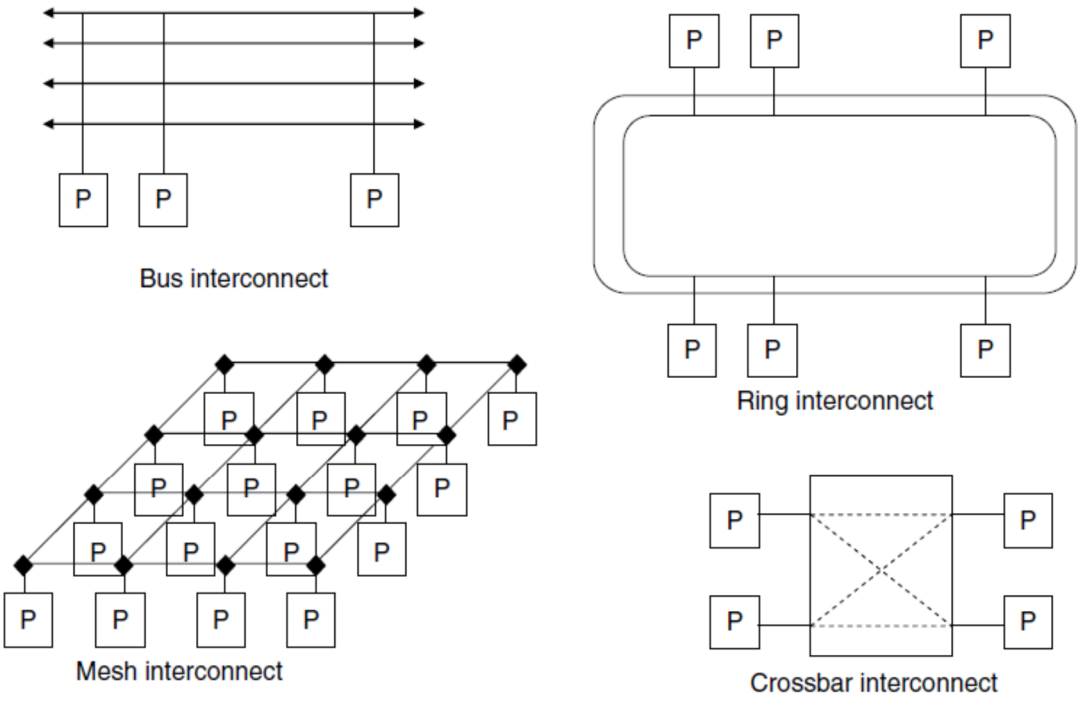

Figure I.06 : Les schémas d'interconnexion des cœurs d'un microprocesseur [15].

Ces derniers temps, les efforts de recherche se sont tournés vers des interconnexions en réseaux optiques sur puce. Car celles-ci :

- Offrent des vitesses plus élevées ;
- Ont une faible consommation électrique ;
- Occupent moins d'espaces ;
- Ont la possibilité de couvrir plusieurs problèmes d'interconnexion [15].

I.D.2) Les multicœurs homogènes/hétérogènes :

Microprocesseur multicœur homogène : Tous les cœurs du microprocesseur sont identiques. Ce qui facilite grandement la programmation, puisqu'il n'est pas nécessaire d'affecter un traitement spécifique à un cœur particulier. C'est pour cette raison qu'actuellement la quasi-majorité des microprocesseurs multicœurs présents sur le marché épousent parfaitement ce paradigme [02,13,15].

Microprocesseur multicœur hétérogène : Le microprocesseur comprend des cœurs différents et spécialisés dans des domaines bien précis (audio, affichage, calcul pur, …etc.). Leur organisation typique comprend un cœur de contrôle qui gère plusieurs cœurs auxiliaires [02,13].

Le processeur Cell conçu par IBM, Sony et Toshiba est l'un des exemples les plus récents de microprocesseurs multicœurs hétérogènes. Il est composé d'un cœur de contrôle nommé PPE « *Power Processor Element* » et de 8 cœurs auxiliaires appelés SPE « *Synergestic Processor Element* », le tout relié par un bus en anneau intitulé EIB « *Element Interconnect Bus* » [02,13,15].

La figure I.07 représente l'architecture du microprocesseur Cell :

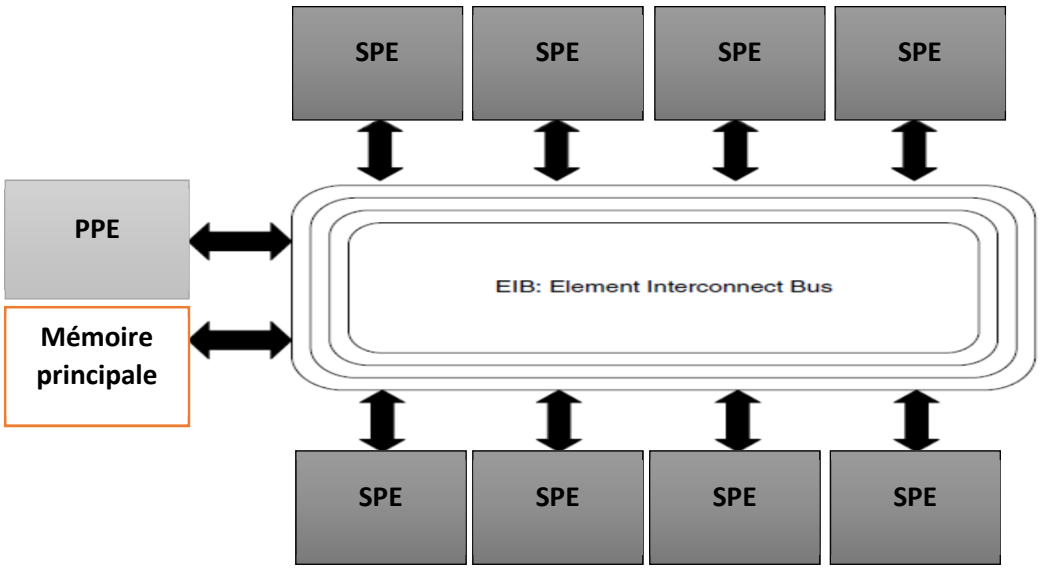

Figure I.07 : Architecture du microprocesseur Cell [15].

I.D.3) L'organisation des caches dans les multi-cœurs :

L'organisation des caches doit être adaptée à la présence de plusieurs cœurs [13].

Deux organisations distinctes sont alors adoptées :

1) **Les caches dédiés :** chaque cœur possède son propre cache, que lui seul peut utiliser [15].
   Avantages :
   - Un temps d'accès réduit ;
   - Un programme exécuté sur un cœur ne pollue pas le cache d'un autre cœur [13].

   Inconvénients :
   - Faible capacité, ce qui est problématique lors d'exécution de programmes ayant besoin de beaucoup d'espace mémoire ;
   - Dans le cas ou 2 cœurs utilisent les mêmes données, celles-ci seront présentes sur les 2 caches de chaque cœur. Ce qui signifie, d'une part, une perte d'espace mémoire, et d'autre part, si l'un des cœurs apporte une modification aux données partagées, il doit en avertir l'autre cœur, d'où la nécessité d'utiliser un protocole de cohérence des caches [13,15].

2) **Les caches partagés :** plusieurs cœurs peuvent se partager dynamiquement un seul cache de façon concurrente. Ces caches sont des mémoires multiports à large bande passante pour permettre à plusieurs cœurs d'y accéder simultanément [13,15].
   Avantages :
   - Ont une plus grande capacité que leurs homologues dédiés ;
   - Évitent la réplication des données partagées entre plusieurs cœurs [13].

   Inconvénients :
   - Ont un temps d'accès plus conséquent, causé par leurs grandes tailles, leurs larges bandes passantes ainsi que leurs natures multiports;
   - Induisent plusieurs cœurs à se marcher sur les pieds [13].

Dans la réalité, compte tenu de ces contraintes. Les caches qui ont besoin d'une faible latence[1] (cache L1) sont des caches dédiés, tandis que les autres sont partagés (L2, L3) [13].

La figure I.08 représente un exemple d'organisation des caches adaptée à la présence de 4 cœurs :

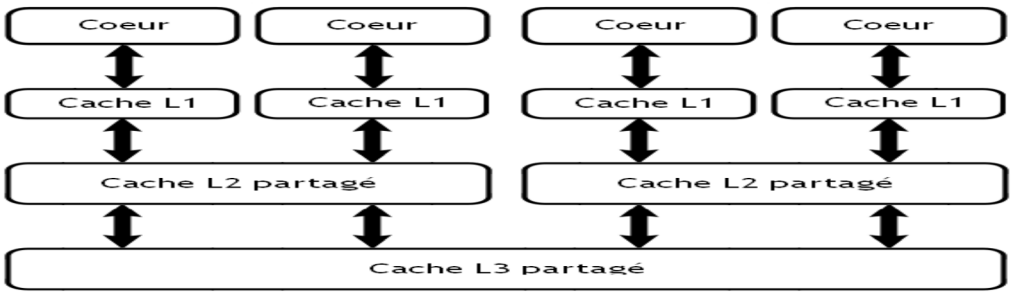

Figure I.08 : Exemple d'organisation des caches sur 4 cœurs.

[1] La durée minimale nécessaire à un cœur pour y récupérer une donnée.

I.D.4) L'accès à la mémoire principale par les multicœurs :

Il existe dans le monde des multicœurs 2 façons d'accéder à la mémoire principale :

a) Accès UMA « *Uniform Memory Access* » (Accès uniforme à la mémoire) ;
b) Accès NUMA « *Non-Uniform Memory Access* » (Accès non-uniforme).

I.D.4.a) UMA :

Ces microprocesseurs sont ainsi nommés, car les temps d'accès à la mémoire principale sont les mêmes quel que soit le cœur demandant l'accès.

Le bus mémoire constitue le seul point d'accès de tous les cœurs vers la mémoire. Lorsque le nombre de cœurs augmente, ce bus peut vite devenir un goulot d'étranglement. Car chaque nouveau cœur envoie des requêtes vers la mémoire principale, la surchargeant d'autant plus [10].

La figure I.09 représente une architecture multicœur avec accès UMA à la mémoire principale :

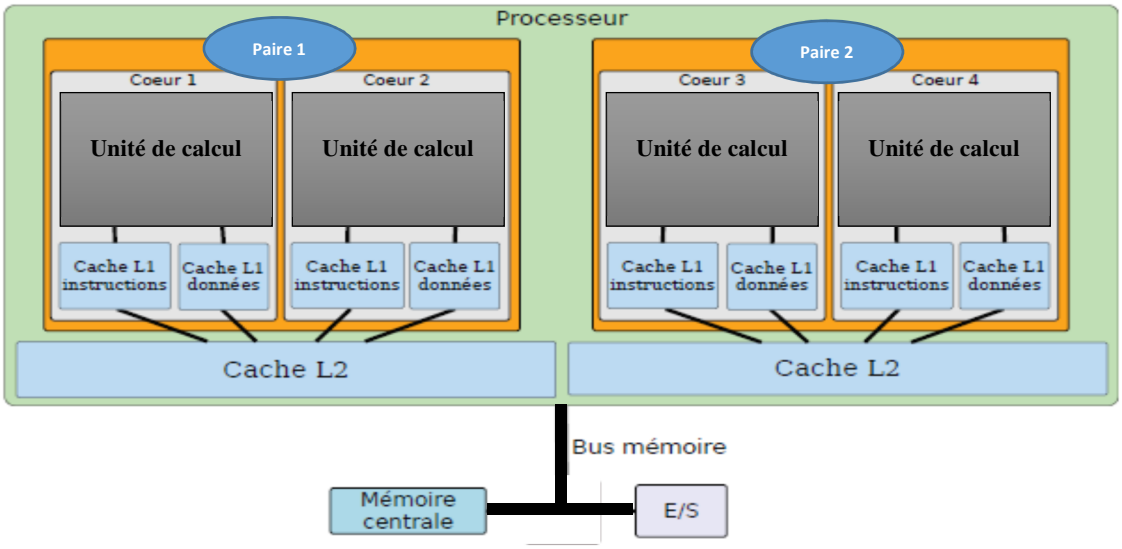

Figure I.09 : Architecture multicœur avec accès UMA à la mémoire principale [10].

I.D.4.b) NUMA :

Les microprocesseurs UMA ont dans leur architecture un goulot d'étranglement : le bus mémoire. C'est pour répondre à ce constat que les fabricants ont mis au point les microprocesseurs NUMA. L'idée est de distribuer la mémoire en plusieurs bancs. Chaque banc est associé à un nœud NUMA. Chaque nœud NUMA peut regrouper plusieurs cœurs.

Malgré la distribution physique de la mémoire, celle-ci apparaît tout de même de façon centralisée au programmeur. Il est donc nécessaire que chaque cœur puisse accéder à toute la mémoire de la machine. Un cœur accède normalement au banc mémoire de son nœud. Lorsqu'en revanche il doit accéder à un nœud distant, un mécanisme de routage se met en place.

En effet, la mémoire du nœud destination n'est pas nécessairement accessible en un seul saut. Les nœuds NUMA sont donc en charge de router les requêtes mémoires des différents cœurs. Chaque saut de nœud a un coût non négligeable, qui s'ajoute à la latence mémoire. C'est en cela que les accès mémoires ne sont pas uniformes [10].

La figure I.10 montre un exemple de microprocesseur NUMA à 16 cœurs, tiré d'une machine basée sur l'architecture Shanghai d'AMD :

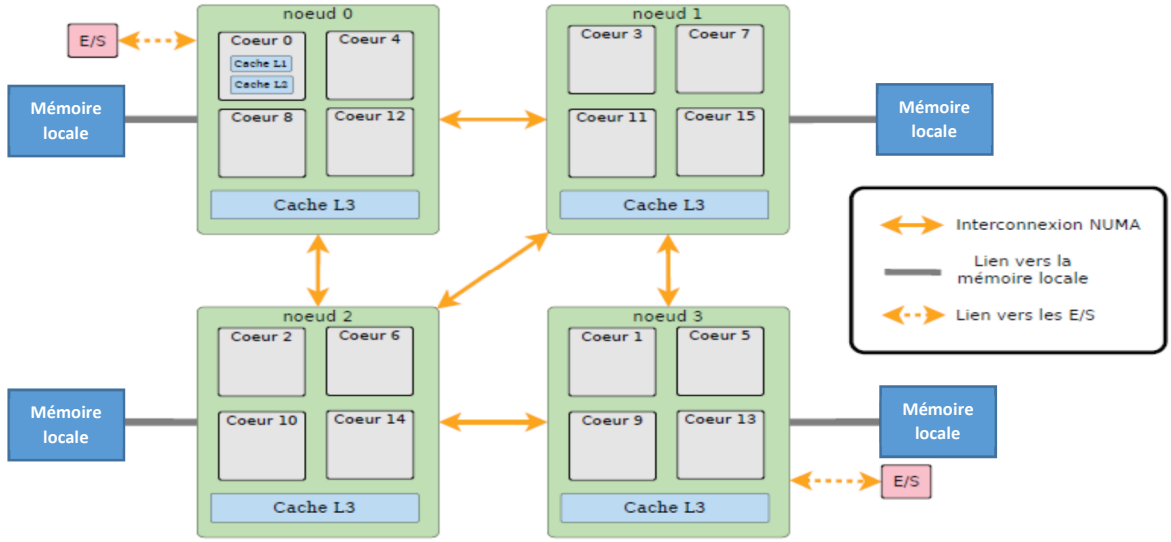

Figure I.10 : Architecture multicœur avec accès NUMA à la mémoire principale [10].

I.D.5) Programmation pour les multicœurs :

Plusieurs programmes séquentiels : La présence de plusieurs cœurs sur un microprocesseur est utile lorsqu'on exécute plusieurs programmes séquentiels en même temps. Car chaque programme sera exécuté par un cœur [14].

Un seul programme séquentiel : Dans le cas où, on ne lance qu'un seul programme séquentiel sur un microprocesseur multicœur, le programme ne bénéficiera pas automatiquement[1] des performances de cette architecture. Car, il n'est exécuté que par un seul cœur, or, il faudrait qu'il soit exécuté par un maximum de cœurs du microprocesseur.

Conclusion, un programme séquentiel n'est pas optimal pour les multicœurs.

Exemple : Un programme utilisant 100 % des performances d'un microprocesseur monocœur, n'utilisera au maximum que 50% des performances d'un bicœur et moins de 6% de celles d'un 16-cœur [16].

Sachant que la programmation parallèle bénéficie du passage à plusieurs cœurs.

[1] Contrairement à la montée en fréquence des microprocesseurs.

Alors, afin qu'un programme séquentiel puisse exploiter tous les cœurs disponibles, il doit être parallélisé[1] [16].

Pour cette raison, et comme les applications serveurs étaient déjà parallèles au moment de l'avènement des multicœurs, les fabricants de microprocesseurs ont considéré que le marché des serveurs était le plus prometteur dans un premier temps, comme le souligne Bernard SEITE, responsable technologie chez AMD : « Toutes les applications serveurs sont déjà prévues pour fonctionner avec plusieurs processeurs, ce qui n'est pas le cas de tous les logiciels pour PC de bureau. » [12].

Mais encore, il est aussi important d'ajouter que dans le monde de Linux près de 99.8% des logiciels sont portés sous plusieurs architectures, ils adoptent eux aussi le parallélisme pour tirer profit des multicœurs [13].

I.E) Parallélisme :

Le parallélisme est le traitement simultané de plusieurs tâches [05].

I.E.1) Le principe du parallélisme :

Un programme parallèle est divisé en plusieurs tâches séquentielles s'exécutant simultanément, habituellement appelées processus [17].

Certains processus sont dits légers, les anglophones utilisant plutôt le mot « *Thread* », qui peut se traduire par fil d'exécution [17].

En fin de compte, le parallélisme consiste en :

1) Chaque programme est divisé en plusieurs tâches ;
2) Chaque tâche est affectée à un thread ;
3) Chaque thread est exécuté par un cœur [18].

Et comme, on dispose d'un multicœur, alors les threads sont exécutés simultanément « en parallèle » [18].

I.E.2) Les types du parallélisme :

Il s'agit ici de présenter les différentes façons possibles de diviser un programme en plusieurs tâches.

[1] Rendu parallèle.

Sachant qu'un programme est une suite d'instructions à effectuer sur un ensemble de données. Alors, soit on fixe[1] les données en divisant les instructions sur les tâches, ou encore, on fixe les instructions en divisant l'ensemble des données sur les tâches.

I.E.2.a) Parallélisme de contrôle :

Ce type de parallélisme reprend la 1re alternative citée plus haut. Toutefois, il est indispensable de garantir l'indépendance[2] entre les séquences d'instruction générées [19].

La figure I.11 illustre ce type de parallélisation :

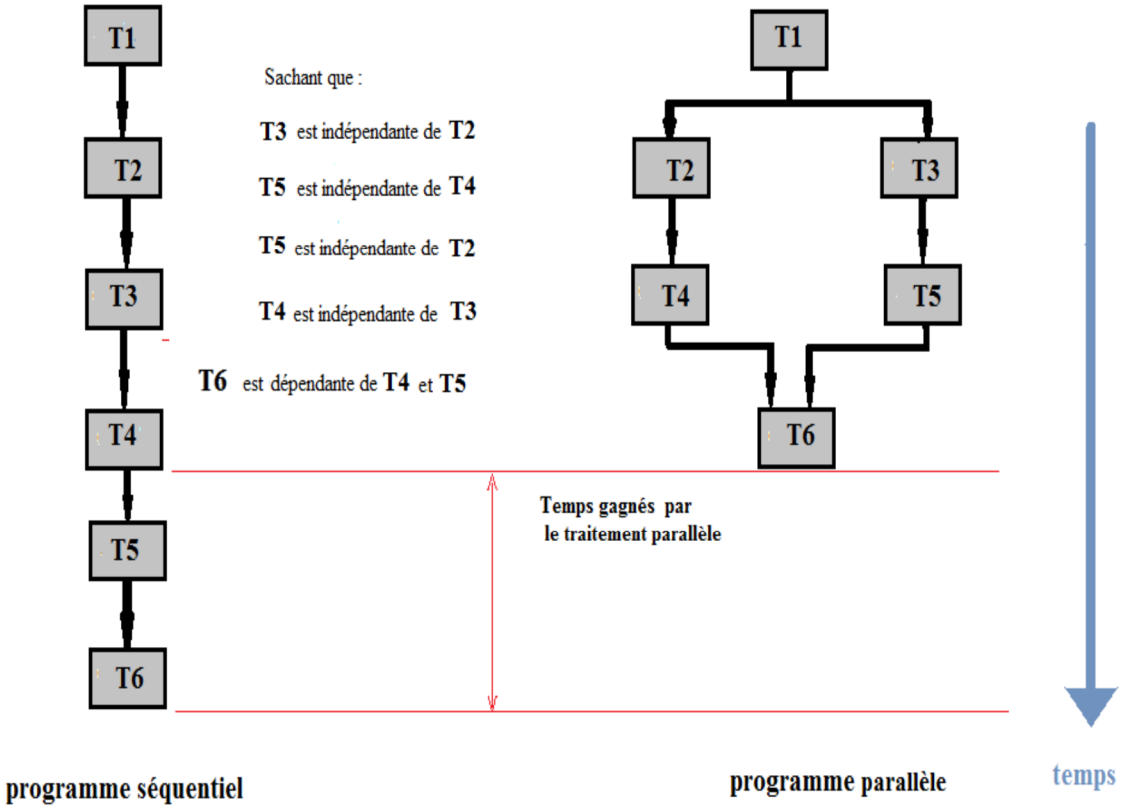

Figure I.11 : Parallélisme de contrôle.

I.E.2.b) Parallélisme de données :

Si un programme utilise un grand nombre de données similaires (vecteurs, matrices, …etc.) sur lesquelles il effectue le même traitement, alors la structure est décomposée puis ses parties sont réparties sur les tâches, lesquels leur effectue le même traitement [20].

[1] Attribut la même donnée à toutes les tâches.
[2] Cette notion sera expliquée plus loin.

Il existe 2 manières de se faire :

Décomposition par blocs : (Distribution statique des données) avant le début des traitements, le nombre de données N est divisé par le nombre de tâches T. Ce rapport (N/T) est appelé bloc de données. Chaque tâche recevra un seul bloc pour toute la durée du traitement. Une fois l'affectation des données faite, on lance les threads [20].

La figure I.12 illustre le fonctionnement de la décomposition par blocs des données :

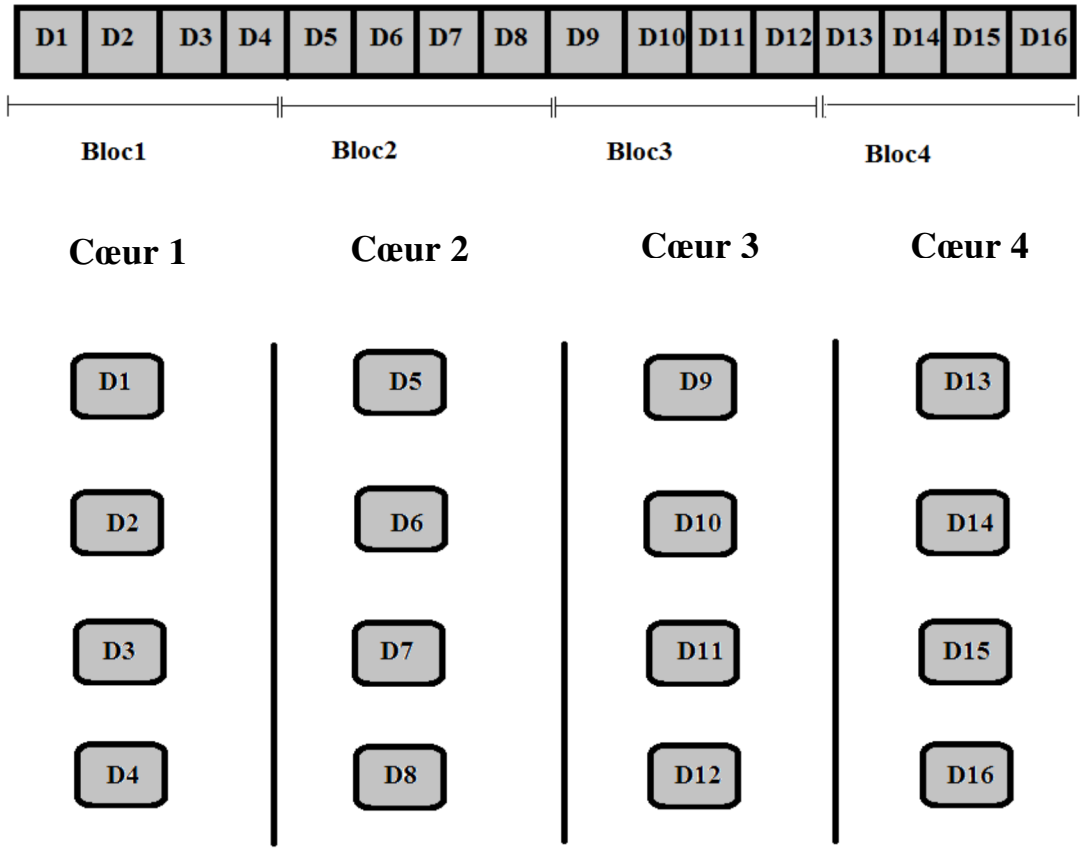

Figure I.12 : Décomposition par blocs des données.

Décomposition cyclique : (Distribution dynamique des données) avant le début des traitements, on précise la taille d'un bloc de données à distribuer. Puis, on affecte pour chaque tâche un seul bloc. Une fois les threads lancés, durant l'exécution, les blocs restants leur seront distribués de façon cyclique, de sorte qu'un thread ayant achevé son traitement sur un bloc se verra attribuer un autre bloc[1] pour boucler dessus avec le même traitement [20].

[1] Le bloc N° (n+x), avec n : le nombre de threads lancés et x : le numéro du bloc précédemment traité par ce même thread.

La figure I.13 illustre le fonctionnement de la décomposition cyclique des données :

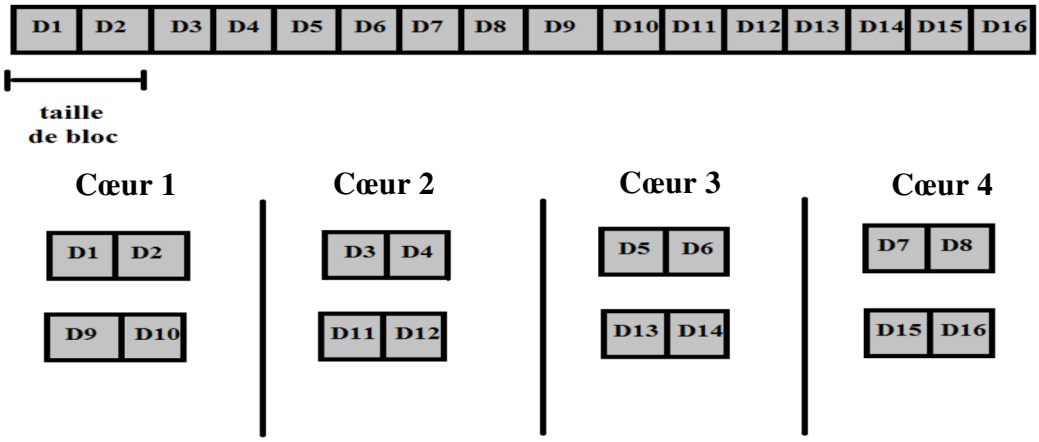

Figure I.13 : Décomposition cyclique des données.

I.E.3) Conception d'un programme parallèle :

La conception d'un programme parallèle pour résoudre un problème donné obéit à une démarche rigoureuse, car l'adoption du parallélisme nous contraint à prendre en considération de nouveaux facteurs, qui compliquent la tâche du programmeur.

Les étapes de cette démarche sont :

a) Étude de la faisabilité & pertinence ;
b) Décomposition ;
c) Synchronisation.

I.E.3.a) Étude de la faisabilité & pertinence :

Faisabilité :

L'étude de la faisabilité consiste à répondre à la question : « Le problème est-il parallélisable ? ». Pour y répondre, le programmeur devrait essayer de déceler « des dépendances de données ».

Il y a dépendance de donnée lorsque, le résultat d'une tâche est le paramètre d'une autre tâche. Ce qui implique que cette dernière ne peut être exécutée qu'après terminaison de la première.

Si ces dépendances sont présentes en grand nombre, alors nul besoin de résoudre le problème avec un programme parallèle. Un programme séquentiel ferait très bien l'affaire, surtout qu'il s'avère être plus simple à implémenter.

À contrario, lorsque le traitement sur une partie des données est indépendant du traitement sur les autres parties, ou alors, lorsqu'il existe des tâches qui peuvent être traitées indépendamment des autres, une solution parallèle donnerait entière satisfaction [21].

Pertinence :

L'étude de la pertinence consiste à répondre à la question : « Une solution parallèle peut-elle apporter un gain de performance ? ». Pour y répondre, rien de mieux qu'une expérimentation avant déploiement [21].

I.E.3.b) Décomposition :

Le problème doit être décomposé selon le type du parallélisme implémentable : parallélisme de contrôle ou de données [20].

Toutefois, il faut veiller à assurer « l'équilibrage des charges », en d'autres termes, une distribution équitable des charges de travail sur les différentes tâches du programme, car dans le cas contraire, la performance globale serait celle de la tâche la plus chargée [20].

Lorsque les tâches effectuent le même traitement, l'équilibrage des charges est trivial, il suffit d'attribuer aux tâches la même quantité de données. Si par contre les tâches exécutent un code différent, il est très difficile d'aspirer à l'équilibrage des charges tant il faut plutôt se soucier de l'indépendance des taches à exécuter en parallèle [20].

I.E.3.c) Synchronisation :

Les tâches dépendantes ont besoin de se synchroniser, soit pendant un échange de données « communication », soit à la suite d'une opération collective « RDV ». Parmi les opérateurs de synchronisation les plus utilisés, on trouve les barrières & les sémaphores.

- Une barrière est un point de RDV. Pour la passer, il faut que tous les threads du programme aient atteint cette barrière ;
- Un sémaphore est utilisé pour protéger une donnée ou une section critique du code. Il permet le passage à la donnée ou section critique pour un nombre fixe de threads.

Conclusion :

Dans ce 1er chapitre, nous avons réalisé que la course aux gigahertz s'est estompée, dès les débuts du 21e siècle, pour laisser place au paradigme des multicœurs mieux loti face aux difficultés de dissipation thermique. Ces derniers nous imposent la programmation parallèle pour tirer un profit maximal de leurs performances, laquelle reste toutefois difficile à mettre en œuvre sans obéir à une démarche assez rigoureuse.

Chapitre II : Traitement d'images et extraction de contours.

Dans ce chapitre, nous allons aborder le traitement d'images en commençant par présenter quelques notions générales liées à l'imagerie. Par la suite, nous allons présenter une des primitives fondamentales dans la reconnaissance des formes qui est la détection des contours, et ce, en étudiant l'algorithme de R.DERICHE développé à cette fin.

II.A) Concepts généraux :

II.A.1) Définition de l'image :

Une image est la représentation d'une scène par la photographie, le film, les arts graphiques ou plastiques.

Une image est un signal bidimensionnel de nature continue. C'est pourquoi elle doit d'abord subir une numérisation avant de pouvoir être traitée sur ordinateur.

Le passage vers une représentation numérique de l'image se fait par deux opérations:

1) **La discrétisation de l'espace :** nommée « l'échantillonnage », qui permet de savoir le nombre de points discrets que contiendra l'image ;
2) **La discrétisation des couleurs :** qu'on appelle « la quantification » qui permet de définir la manière de représentation des couleurs ainsi que le nombre de celles-ci.

Le résultat de ces opérations est un ensemble structuré de données appelé « image numérique ». Cette structure est en fait une matrice composée de « *pixels* » [22].

En considérant la palette des couleurs utilisée dans une image numérique on peut dire qu'elle est :

- En couleur ;
- En noir et blanc ;
- Ou en niveau de gris.

II.A.2) Définition d'un pixel :

Étymologiquement, le mot anglo-saxon « *pixel* » est la fusion des deux mots « *picture* » (image) et « *element* » (élément).

Le pixel est la donnée la plus élémentaire qui constitue une image. Il est caractérisé par des coordonnées (x,y)[1] déterminant sa position dans l'image et une valeur déterminant sa luminance.

La figure II.01 montre un exemple d'un pixel p de coordonnées (i,j) dans une image ayant NC colonnes et NL lignes :

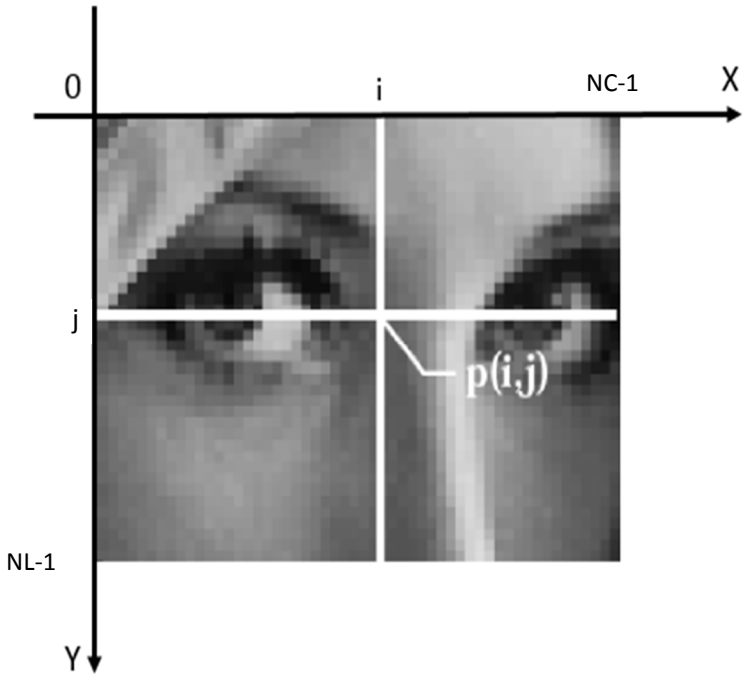

Figure II.01 : Exemple de pixels sur une image numérique [22].

[1] En considérant les images en 2D.

Codage des valeurs d'un pixel :

Les valeurs d'un pixel varient selon le type de l'image. Soit un pixel dans une image :

- **En couleurs :** sa valeur est vectorielle et prend trois composantes (R,V,B) dans l'espace des couleurs primaires (Rouge,Vert,Bleu). Chacune de ces composantes (codée sur 1octet) prend une valeur entière allant de 0 à 255 représentant sa dose dans la composition de la couleur du pixel [02,22].
- **En noir et blanc :** sa valeur est binaire soit (0) pour le noir ou (1) pour le blanc [22].
- **En niveau de gris :** la valeur du pixel est un scalaire codé sur 1octet qui représente sa luminosité elle va du noir absolu (0) au blanc parfait (255) [02,22].

Voisinage d'un pixel :

Le voisinage d'un pixel est une relation géométrique. Un pixel de coordonnées (x,y) partage une arrête avec 4voisins : (x,y+1), (x,y-1),(x+1,y) et (x-1,y), on parle alors de connexité 4. Comme il partage un sommet avec 8voisins : (x-1,y+1), (x,y+1),(x+1,y+1),(x-1,y),(x+1,y),(x-1,y-1),(x,y-1),(x+1,y-1) et on parle alors de connexité 8 [02].

La figure II.02 montre une illustration de cette relation :

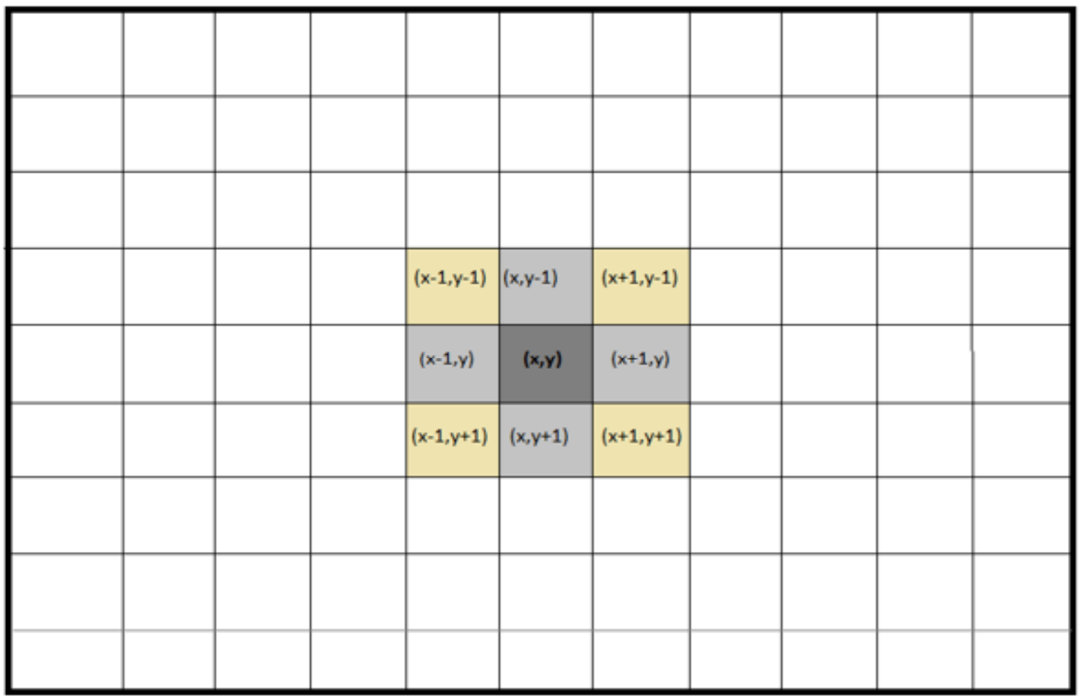

Figure II.02 : Voisinage d'un pixel.

II.A.3) Propriétés d'une image numérique:

II.A.3.a) La dimension :

Comme définie plus haut, une image est une matrice de pixels. Sa dimension est donnée par le nombre de colonnes et le nombre de lignes qui la composent, leur produit est alors égal au nombre de pixels qui constitue l'image [22].

Sur la figure II.03, on note la différence marquée sur la même image selon une variation de dimension, allant de 256x256 soit 65536 pixels dans l'image la plus à gauche, à 32x32 soit 1024 pixels sur l'image la plus à droite :

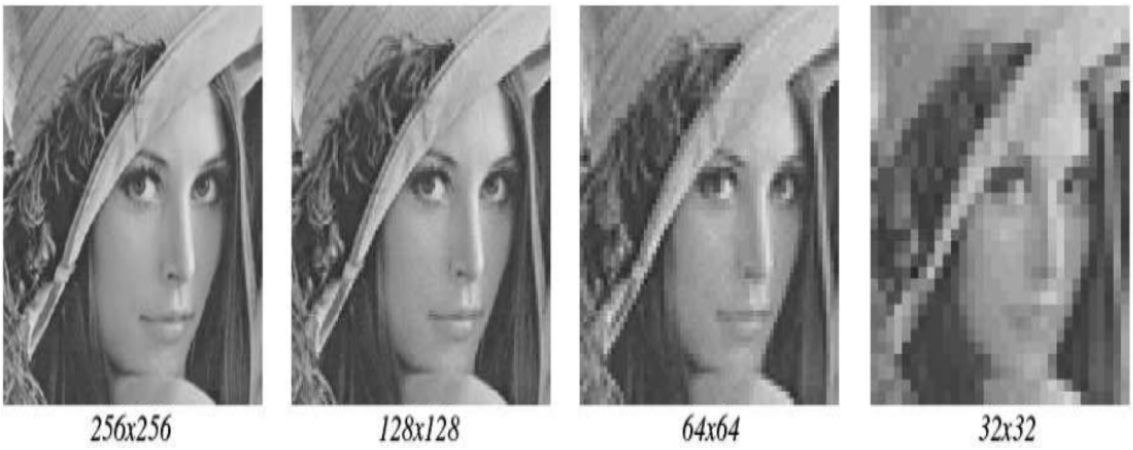

Figure II.03 Différentes dimensions d'une image [22].

II.A.3.b) Le bruit :

Comme l'image est un signal, elle peut être altérée. Le bruit est une forme d'altération de l'image. C'est un signal parasite qui s'ajoute de manière aléatoire aux détails de l'image numérisée. 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 [02,22].

La figure II.04 montre la différence entre une image bruitée (à droite) et une image non bruitée (à gauche) :

Figure II.04 : Image bruitée et image non bruitée [22].

Le problème du bruit dans les images peut être résolu en utilisant des filtres.

II.A.4) Le filtrage :

Les filtres sont des opérateurs qui éliminent des éléments perturbateurs de l'image afin d'améliorer sa visualisation ou la préparer à un traitement postérieur.

Il existe plusieurs sortes de filtres, dans ce qui suit nous allons nous intéresser aux filtres linéaires. Ils s'expriment sous forme d'un produit de 2 matrices, la matrice image et « un masque de convolution ».

Un masque de convolution est généralement une matrice carrée[1] de dimension impaire.

Soit R(x,y) un pixel de l'image résultante R du filtrage de l'image I par un masque de convolution K, comme exprimé par la formule suivante :

$$R(x,y) = \sum_{u=-n}^{u=n} \sum_{v=-n}^{v=n} I(x+u, y+v) \times K(u+n, v+n) \qquad (1)$$

Avec : $n = \frac{(dimension\ K)-1}{2}$

Exemple: on applique un produit de convolution au pixel (2,2) de la matrice en figure II.03 par un noyau K(3,3) :

(150×5)+(−1×100)+(−1×100)+(−1×100)+(−1×100) = 350.

[1] Nombre de lignes=nombre de colonnes.

La figure II.05 est une représentation graphique de cet exemple :

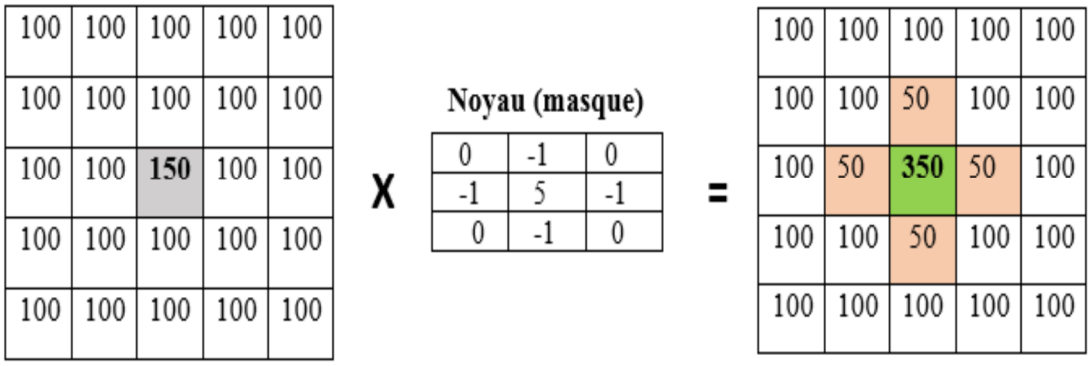

Figure II.05 : Exemple de filtrage sur le pixel central.

Dès que l'on cherche à appliquer cet opérateur sur toute la matrice un problème se pose subséquemment, les pixels des bords de l'image ne peuvent pas être traités. Plusieurs solutions ont été envisagées :

- Ne pas traiter les pixels des bordures ;
- L'extérieur de l'image est composé de pixels de couleur unie ;
- Les bords sont dupliqués à l'extérieur de l'image ;
- Ou alors, l'image est répétée en cycle [22].

Nous allons maintenant nous intéresser à une classification des filtres linéaires, qui les regroupe en 3 catégories :

1) Filtre passe-haut :

C'est un filtre qui laisse passer les hautes fréquences et qui atténue les basses fréquences relativement à une fréquence de coupure.

2) Filtre passe-bas :

Contrairement au filtre passe-haut un filtre passe-bas est un filtre qui laisse passer les basses fréquences et qui atténue les hautes fréquences supérieures à la fréquence de coupure.

3) Filtre passe-bande :

Un filtre passe-bande est un filtre qui ne laisse passer qu'une bande ou un intervalle de fréquences comprises entre une fréquence de coupure basse et une fréquence de coupure haute.

II.B) Extraction de contours :

Une grande part de l'information d'une image est portée sur ses contours. Il est donc intéressant de procéder à leur extraction pour pouvoir sauvegarder le maximum d'informations que contient une image [23].

II.B.1) Qu'est-ce qu'un contour ?

Un contour est une frontière entre 2 objets sur une image. Il est traduit par une variation brusque de l'intensité lumineuse [24].

La figure II.06 montre une image et la courbe qui représente l'intensité lumineuse d'une de ses lignes qui présente un contour. Sur la courbe, le contour correspond au changement rapide de l'intensité lumineuse :

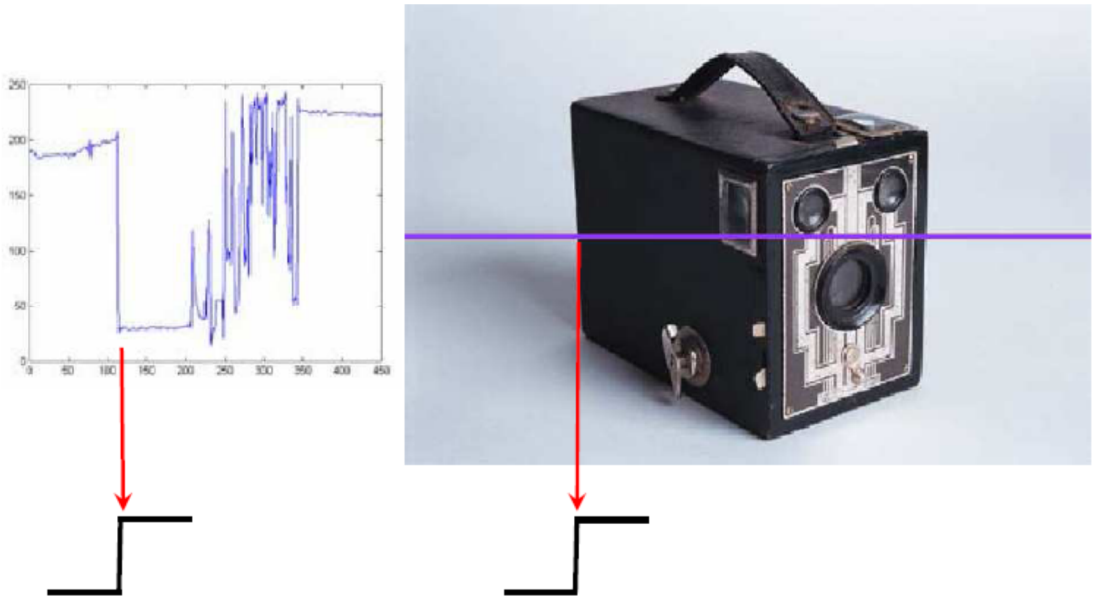

Figure II.06 : Contour dans une image [24].

Toutefois, il est important de souligner qu'un contour n'est pas nécessairement situé à la frontière entre deux objets, on peut notamment le remarquer sur la figure II.07 :

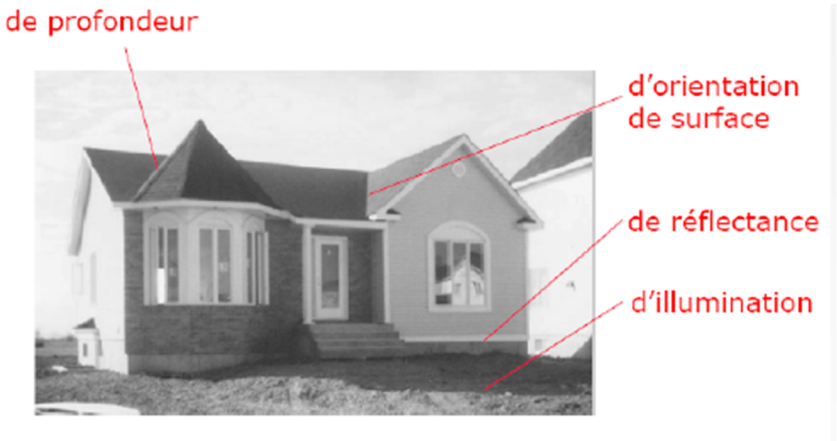

Figure II.07 : Image avec des contours autres qu'une frontière entre 2 objets [24].

En effet, il existe plusieurs sortes de contours. La figure II.08 en présente 3 :

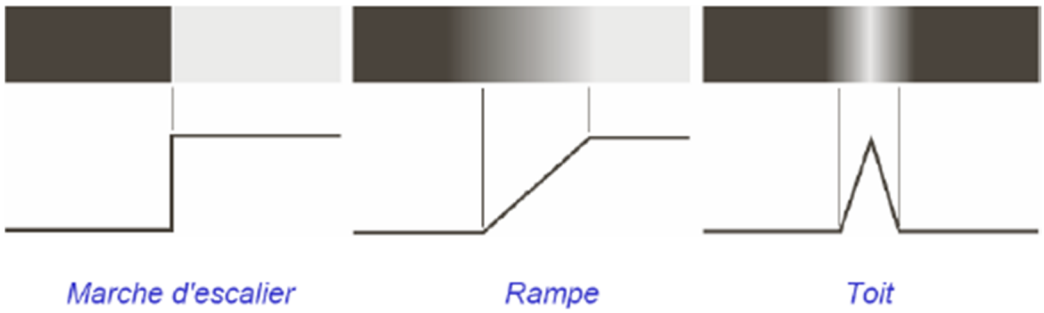

Figure II.08 : Types de contours [24].

II.B.2) Domaines d'application de l'extraction des contours :

L'extraction des contours est utilisée dans divers domaines usants des images, parmi lesquels on peut citer :

- **Le domaine médical :** l'utilisation des contours dans les scanographies permet d'identifier les tumeurs cancéreuses.
- **L'astronomie :** les contours dans les images satellitaires facilitent la reconnaissance des corps célestes et la détection des nouveaux corps.

II.B.3) Approches pour l'extraction de contours :

II.B.3.a) Approche naïve :

L'extraction de contours est un problème qui n'a pas de solution générale, il existe cependant plusieurs approches permettant de le résoudre [22].

Pour mieux appréhender ce problème, nous allons commencer par étudier une solution simple. Pour ce faire, on considère l'image présentée sur la figure II.09. C'est une image non bruitée et constituée de pixels noirs et blancs uniquement :

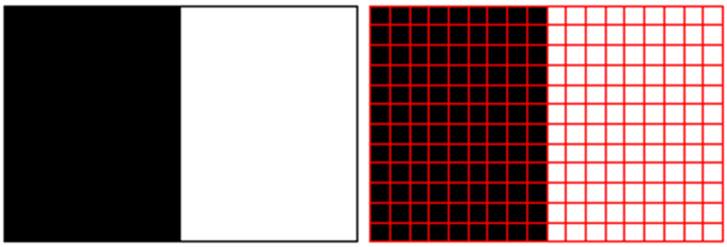

Figure II.09 : Image naïve pour comprendre l'extraction de contours [22].

On considère alors que les contours sur cette image sont les pixels blancs dont le voisin de gauche est noir. De plus, les pixels des bords de l'image ne seront pas traités.

On obtient l'image des contours en procédant comme suit :

Soit :

- Ic[i,j] un pixel de l'image des contours ;
- I[i,j] un pixel de l'image source.

Alors: $Ic[i, j] = I[i, j] - I[i-1, j]$ (2)

La figure II.10 représente deux images et le résultat de l'application de notre détecteur :

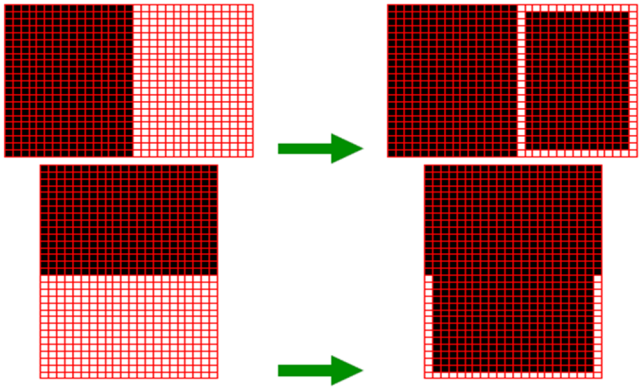

Figure II.10 : Images de test (à droite) et images de contours (à gauche) [22].

Sur la figure II.10, on remarque que sur la deuxième image les contours verticaux ne sont pas détectés. Pour remédier à cela, nous allons améliorer notre détecteur en le composant de 2 détecteurs :

1) Un détecteur horizontal :

$$I_h[i][j] = I[i][j] - I[i-1][j] \qquad (2)$$

2) Un détecteur vertical :

$$I_v[i][j] = I[i][j] - I[i][j-1] \qquad (3)$$

Pour combiner les résultats des deux détecteurs et obtenir un résultat final, nous allons employer la norme euclidienne des deux vecteurs comme suit :

$$I_f[i][j] = \sqrt{(I_h[i][j])^2 + (I_v[i][j])^2} \qquad (4)$$

Essayons maintenant notre détecteur sur une image contenant des contours diagonaux on voit, en figure II.11, cette image et celle de ses contours :

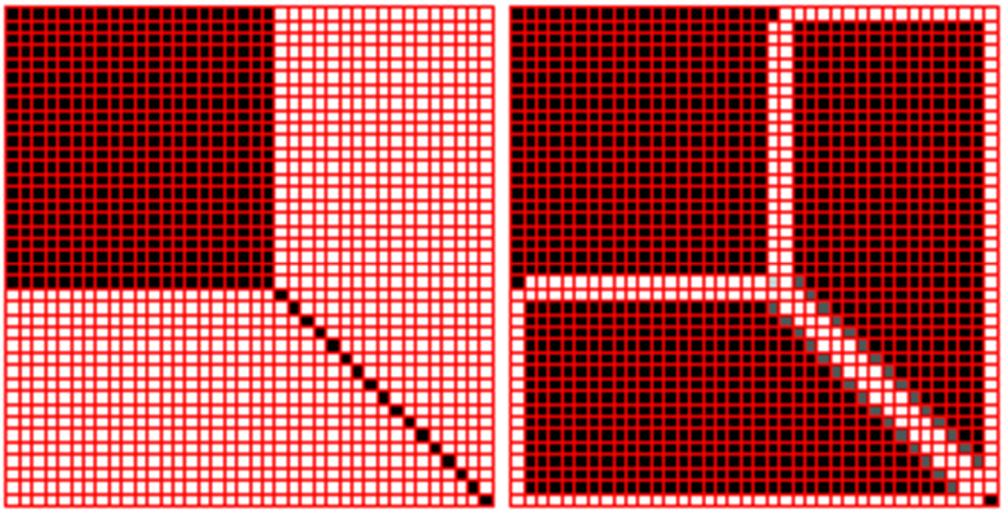

Figure II.11 : Image de contours diagonaux [22].

Le problème est que les contours ont une épaisseur de 2 pixels. On peut alors déduire la difficulté de détection des détails fins.

II.B.3.b) Approches par dérivation :

Comme les contours sont traduits par un changement brusque de l'intensité lumineuse dans une image, si on lui effectue une dérivation ces changements seront tout de suite plus visibles [24].

Sur la figure II.12**,** on voit en bleu la courbe d'une ligne d'une image et en rouge la courbe de sa dérivée :

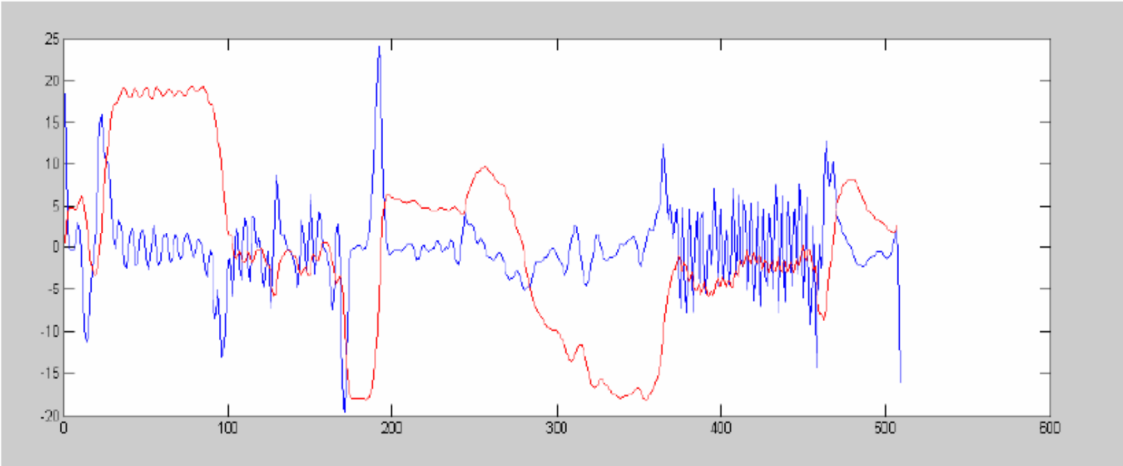

Figure II.12 : courbe d'une ligne d'une image et sa dérivée [24].

En effet, il existe des méthodes dérivatives pour l'extraction des contours et qui sont :

1) **Les dérivées premières :** appelées « gradient », les contours correspondent aux pics de la 1ère dérivée appelés « maxima locaux ».
2) **Les dérivées secondes :** appelées « Le Laplacien », les contours sont les points du passage à zéro de la 2ème dérivée [24].

La figure II.13 représente les méthodes dérivatives pour l'extraction d'un contour :

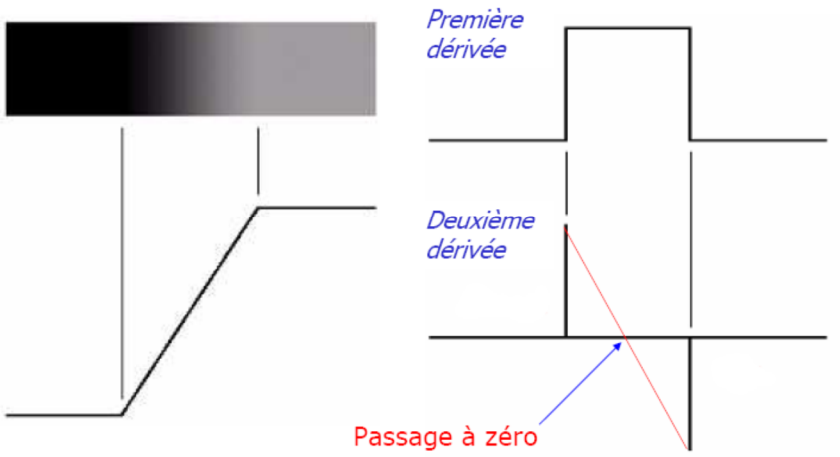

Figure II.13 : Première dérivée et deuxième dérivée d'un contour [24].

Cependant, lorsque l'image est bruitée elle présente un signal sinusoïdal parasite, on suppose qu'il a la forme suivante :

$$bruit = \varepsilon \sin(\omega x) \qquad (5)$$

En effectuant la dérivation, on obtient :

$$bruit' = \varepsilon\omega \cos(\omega x) \qquad (6)$$

On remarque que ce signal s'amplifie, car sa dérivée le multiplie par un facteur ω. Des bruits de hautes fréquences vont alors perturber l'image à grande échelle. Il convient de ce fait d'appliquer des filtres lisseurs[1] qui permettent de réduire l'intensité du bruit présent dans l'image [22].

II.B.4) Critères de qualité d'un détecteur de contours :

En 1986, John F Canny, un professeur à l'université de Californie à Berkeley, a énuméré 3 critères d'optimalité d'un détecteur de contours, lesquels sont :

- **Bonne détection :** Les faux contours dus au bruit doivent être éliminés, cela est possible que lorsque l'image est bien lissée ;
- **Bonne localisation :** La localisation doit être précise, il s'agit de minimiser la distance entre les points détectés et les contours réels ;
- **Unicité de la réponse :** Il arrive parfois que plusieurs contours soient détectés, mais qu'il n'en existe qu'un seul sur l'image réelle, ainsi un filtre doit pouvoir minimiser le nombre de réponses et n'en renvoyer qu'une seule par contour.

[1] Le lissage est une égalisation des valeurs des pixels d'une image.

II.B.5) Présentation du filtre de Deriche :

Le filtre de Deriche, a été conçu en 1987 par Rachid Deriche, directeur de recherche à l'Institut national de recherche en informatique et en automatique « INRIA », pour détecter de manière optimale les contours d'une image en répondant aux critères posés par Canny.

Ce filtre s'applique sur des images en niveaux de gris. Il s'apparente à une succession d'étapes citées ci-dessous :

1) Le lissage :

On applique un filtre de lissage passe-bas à réponse impulsionnelle infinie. L'avantage d'un tel filtre est qu'il peut être adapté aux caractéristiques de l'image à traiter et ce en utilisant un seul paramètre α strictement positif. Ce paramètre α définit le compromis entre la détection et la localisation.

Si la valeur de α est faible (habituellement entre 0,25 et 0,5), il en résulte une meilleure détection.

D'autre part, une meilleure localisation est obtenue lorsque le paramètre α a une valeur plus élevée (environ 2 ou 3). Pour la plupart des cas normaux, la valeur du paramètre est recommandée à 1 [02,22,25].

2) Calcul du gradient :

Permet de ne retenir de l'image traitée que les zones de contraste élevé, c'est-à-dire où l'intensité lumineuse passe de façon brutale, d'une valeur élevée a une valeur faible(ou vice versa) [02].

Dans une première phase, un filtrage récursif est appliqué à chaque ligne de l'image à traiter de gauche à droite et de droite à gauche.

Une seconde phase applique alors au résultat de la 1ère phase le second filtre récursif au niveau de chaque colonne de haut en bas et de bas en haut.

Ce filtre permet d'obtenir le lissage d'une image I, sa dérivée directionnelle en x : *Ix*, ou sa dérivée directionnelle en y : *Iy*, ce qui constitue le gradient de l'image.

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 NG1 du point M1 et NG2 du point M2 [23]. La norme du gradient est donnée par l'expression :

$$NG(i,j) = \sqrt{Ix(i,j)^2 + Iy(i,j)^2}$$

4) Seuillage :

Le rôle de cette étape est de produire une image en noir et blanc à partir de l'image en niveau de gris résultante des étapes précédentes. En procédant comme suit :

- **Calcul du seuil :** Le seuil est égal à la moyenne des niveaux de gris des maxima locaux [23]. Et est calculé comme suit :

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

Ng : le niveau de gris du point maxima local.

M : le nombre de points non nuls dans l'image.

l : le nombre de lignes de l'image.

c : le nombre de colonnes de l'image.

- **Transformation des pixels :** Les pixels dont l'intensité lumineuse est inférieure au seuil deviendront noirs, les autres deviendront blancs, comme suit :

NG(i ,j) = 255 pour N(i ,j) > S

ET

NG(i ,j) =0 pour NG (i ,j) < S

La figure II.14 est un schéma représentant les étapes de l'implémentation du filtre de R.Deriche :

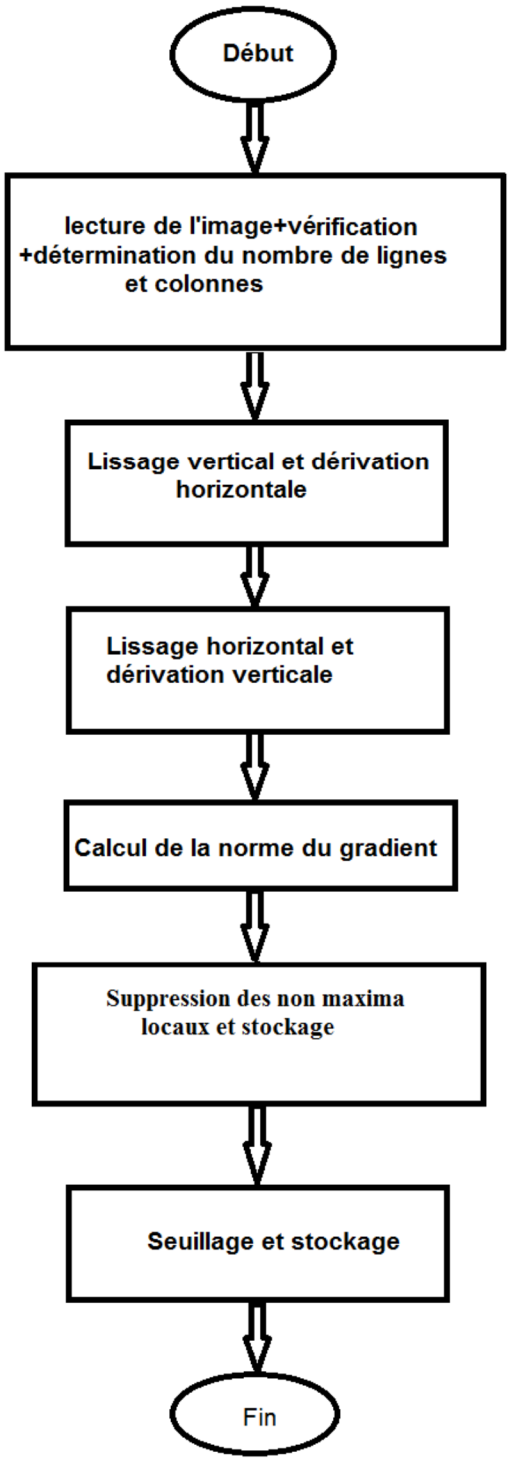

Figure II.14 : Algorithme de R.Deriche.

Conclusion:

Au terme de ce chapitre, nous constatons que l'illustratif algorithme de R.Deriche sied à l'exercice de la parallèlisation dû à son nombre conséquent d'opérations s'appliquant sur des données de nature volumineuse.

La fin de cette première partie « théorique » étant atteinte, nous allons dans ce qui suit passer au volet pratique de notre travail, en présentant dès le chapitre suivant les outils employés pour réaliser notre application dédiée à l'extraction de contours d'images via traitements parallèles.

DEUXIÈME PARTIE : DE LA MÉTHODOLOGIE À L'EXPÉRIMENTATION

Chapitre III : Les outils de développement.

Cette deuxième partie du mémoire est destinée au volet pratique de notre projet.

Nous l'entameront avec ce chapitre consacré à la définition des outils de développement nécessaires à la réalisation de l'application.

Les outils présentés ici se résument au C++ puis Qt.

III.A) C++ :

III.A.1) Présentation du C++ :

Le C++ est un langage de programmation compilé. Il est aussi « de bas niveau », ce qui signifie qu'il est assez proche du code machine [26]. Cette notion étant relative, la figure III.01 situe le niveau langage C++ par rapport à d'autres langages de programmations :

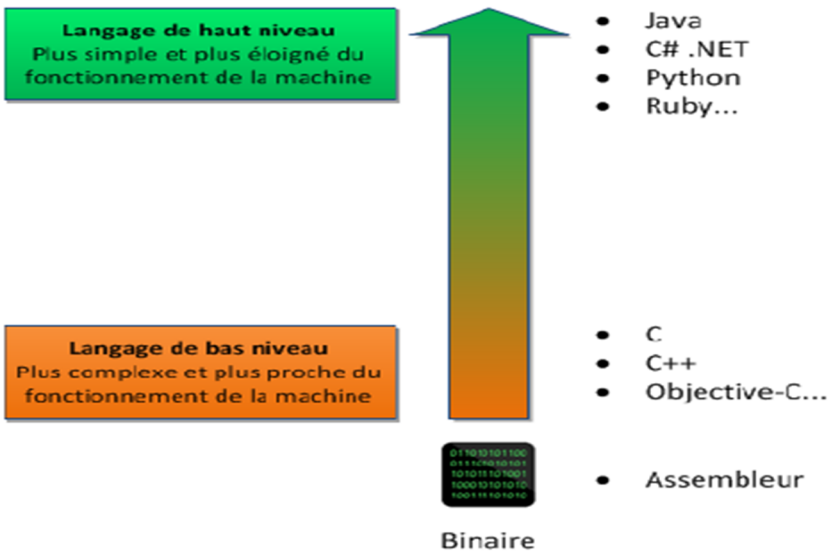

Figure III.01 : Niveaux des langages de programmation [26].

III.A.2) Historique du C++ :

Développé en 1983, par l'informaticien Danois Bjarne STROUSTRUP, dans les laboratoires Bell d'AT&T. Le C++ de son nom signifiant « C incrémenté », est une évolution du langage C. En effet, Bjarne STROUSTRUP, en analysant le noyau UNIX en vue de faire du calcul distribué, jugea qu'il manquait plusieurs fonctionnalités au langage C, commença à l'améliorer y ajoutant les notions d'orienté objet pour en faire le C++ [26-27].

Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/IEC 14882:1998). Le standard a ensuite été amendé par l'erratum technique de 2003 (ISO/IEC 14882:2003). Le standard actuel a été ratifié et publié par l'ISO en septembre 2011 sous le nom de (ISO/CEI 14882:2011) (aussi appelé C++11). Une mise à jour mineure a été publiée en 2014 sous le nom de (ISO/IEC 14882:2014). Le prochain standard devrait être publié en fin septembre 2017 [28-29].

III.A.3) Avantages du C++ :

Le C++ présente les avantages suivants :

III.A.3.a) Bas niveau :

Le C++ permet de développer des programmes ayant plus de contrôle sur le matériel, et ainsi d'obtenir de meilleures performances [26].

III.A.3.b) Très répandu :

Il fait partie des langages de programmation les plus utilisés sur la planète. Ce qui en fait un langage très documenté [26].

III.A.3.c) Rapide :

Ce qui en fait un langage de choix pour les applications critiques qui ont besoin de performances. C'est en particulier le cas du traitement d'image, mais aussi des outils financiers ou de certains programmes qui doivent fonctionner en temps réel [26].

III.A.3.d) Portable :

Un même code source peut être transformé sans problème en exécutable sous les systèmes d'exploitation Windows, Mac OS et Linux, nul besoin de réécrire le programme [26].

III.A.3.e) Extensible :

Il existe de nombreuses bibliothèques[1] pour le C++. Ainsi, en le combinant avec, on peut créer des programmes fenêtrés, 3D, réseaux,… etc [26].

III.A.3.f) Multi-paradigme :

Ce qui signifie qu'on peut créer des programmes C++ en utilisant différentes techniques. On peut alors choisir, selon les besoins, la programmation procédurale « classique » ou opter pour une programmation « orientée objet » [26].

III.A.4) Programmation orientée objet du C++ :

Les langages à objets comme le C++ utilisent des « briques logicielles intelligentes » : les objets. Un programme n'est alors qu'une collection d'objets qui coopèrent [30].

III.A.4. a) Objets :

Un objet représente une entité du monde physique.

En pratique, un objet est une structure regroupant :

- **Des attributs :** caractérisant son état ;
- **Des méthodes :** caractérisant son comportement. Ce sont des fonctions destinées à lire/modifier les valeurs des attributs de l'objet [30].

III.A.4.b) Classes :

Une classe joue le rôle d'un moule à objet, tout objet dérive alors d'une classe, on dit qu'un objet est une « instance » d'une classe.

Une classe est, comme l'objet, décrite par des attributs et des méthodes [30].

- **Les constructeurs :**

  Un constructeur est une méthode appelée à chaque fois qu'on crée un objet.

  Un constructeur est une méthode qui :

  - Porte le même nom que la classe ;
  - Ne retourne aucun résultat, il n'a pas de type de retour [30].

[1] Les bibliothèques sont des extensions pour le langage [04].

- **L'encapsulation :**

En programmation orientée objet, l'utilisateur ne doit pas pouvoir accéder aux attributs d'un objet mais qu'à ses méthodes, ce concept est appelé « encapsulation ». Chaque membre d'une classe, attribut ou méthode, se voit alors attribuer un droit d'accès par les mots clés « *public* » (public) ou « *private* » (privé).

Les méthodes d'une classe doivent être déclarées publiques. Par contre, ses attributs doivent être privés. Pour pouvoir accéder aux attributs privés, en lecture ou en écriture, on utilise des méthodes spéciales appelées « accesseurs ».

L'utilisation des accesseurs en écriture permet de faire des tests pour vérifier la validité des valeurs qu'un utilisateur veut affecter à un attribut, ce qui garantit la cohérence de l'état de l'objet [30].

- **Séparation des prototypes et définitions :**

En pratique, une classe est constituée d'un ensemble de fichiers, cette séparation permet une meilleure manipulation et facilite la modification d'une classe.

Un projet C++ est donc constitué d'un ensemble de fichiers lesquels sont :

  - **Les fichiers d'en-tête :** d'extension « .h » pour « *headers* » (en-tête). Chacun contient la déclaration d'une classe comportant les déclarations des attributs ainsi que les prototypes des méthodes ;
  - **Les fichiers d'implémentation :** d'extension « .cpp » pour « C Plus '+' Plus '+' ». Chacun contient les définitions de toutes les méthodes d'une seule classe ;
  - **Le fichier « main.cpp » :** contient l'implémentation de la fonction « main » qui est le point d'entrée du programme. La création et manipulation des objets se fait dans ce fichier [26].

III.B) Qt :

III.B.1) Présentation de Qt :

Qt (voir son logo en figure III.02) est un « *Framework* » (ensemble de bibliothèques[1]) orienté objet et écrit en C++.

Qt est donc constituée d'un ensemble de bibliothèques, appelées « modules ». On peut y trouver entre autres les modules ci-dessous :

- **Module GUI** : « *Graphical User Interface* » (Interface utilisateur graphique) pour la création de fenêtres ;
- **Module réseau** : pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un lecteur de flux RSS ou un client FTP ;
- **Module SQL** : pour accéder aux bases de données (MySQL, Oracle, PostgreSQL) ;
- **Module programmation parallèle** : pour simplifier le développement d'applications pouvant s'exécuter en parallèle sur tous les cœurs dont dispose le microprocesseur [31-33].

Figure III.02 : Logo du « *framework* » Qt.

III.B.2) Histoire de Qt :

Qt signifie « *Cute* » (Mignonne), parce que ses développeurs trouvaient que la lettre Q était jolie dans l'éditeur de texte Emacs, qu'ils ont eu à utiliser pour développer ce « *Framework* ».

Le développement de Qt a commencé en 1991 par Haavard Nord et Eirik Chambe-Eng au « *Norwegian Institute of Technology* » à Trondheim. Et en 1993, le noyau de Qt est fin prêt. C'est à la fin de cette année-là que Haavard Nord propose de créer une entreprise pour commercialiser le « *Framework* ».

Quasar Technologies est créé le 4 mars 1994 et renommé 6 mois plus tard en Trolltech, puis Qt Software et enfin Qt Development Frameworks qui devint le 9 aout 2012 une filiale de Digia après que Nokia la lui eut cédée pour 5 millions d'euro.

[1] Une bibliothèque est aussi un ensemble de classes [04].

Le 26 mai 1995 est annoncée la première version publique de Qt, puis le 24 septembre 1996 la version 1.0 est publiée. La dernière version de Qt, la 5.0 est sortie le 19 décembre 2012 [34].

III.B.3) Licence de Qt :

Qt est distribué sous deux licences, au choix : LGPL ou propriétaire. Celle qui nous intéresse est la licence LGPL, car elle nous permet d'utiliser gratuitement Qt. On peut aussi bien réaliser avec, des programmes libres[1] que des programmes propriétaires [35].

III.B.4) Avantages de Qt :

Qt présente les avantages suivants :

III.B.4.a) Multi-langage :

Qt est conçu pour être utilisé en C++. Toutefois, il est aujourd'hui possible de l'utiliser avec d'autres langages comme Java ou Python, en y incorporant des « *bindings* »[2] [31-32].

III.B.4.b) Très bien documenté :

La documentation de Qt est facile à utiliser, elle détaille toutes les fonctionnalités de Qt, même les plus récentes. Il existe deux moyens d'y accéder :

- **Avec accès à Internet :** Sur le site « doc.qt.io/ »;
- **Sans accès à Internet :** Sur le navigateur « Qt Assistant », installé avec Qt [31-32]. Voir la figure III.03 :

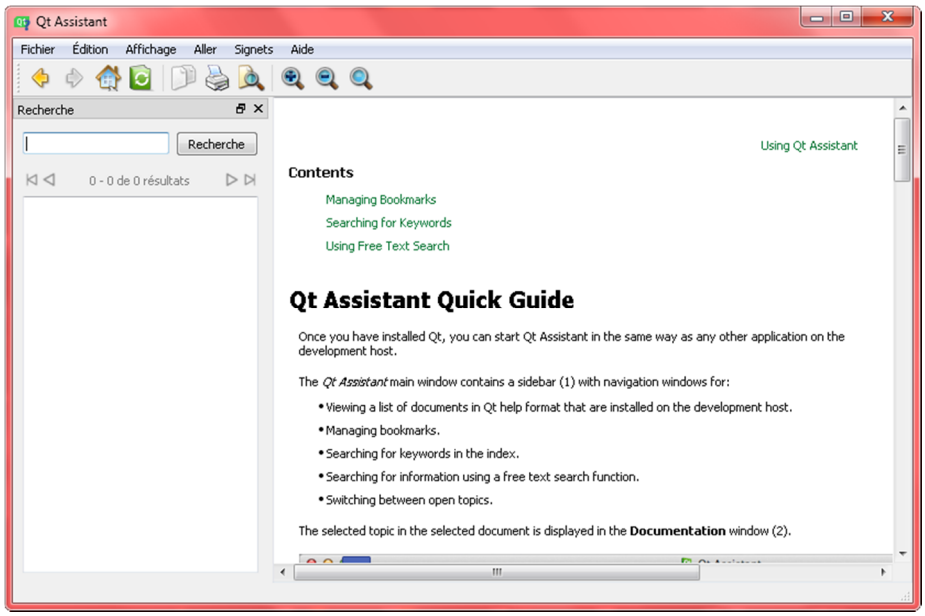

Figure III.03 : Qt Assistant.

[1] Programmes dont le code source est public et, dont on autorise la modification par d'autres personnes [04].
[2] Binder : est le fait de lier [04].

III.B.4.c) Multiplateforme :

Grâce à cette technique, les fenêtres qu'on code ont une apparence adaptée à chaque OS « *Operating System* » (Système d'Exploitation). On code pour Qt et Qt traduit les instructions pour l'OS.

En voici une démonstration. La figure III.04 représente la même fenêtre créée avec Qt, mais sous différents OS (De gauche à droite : Windows 7, Mac OS X) :

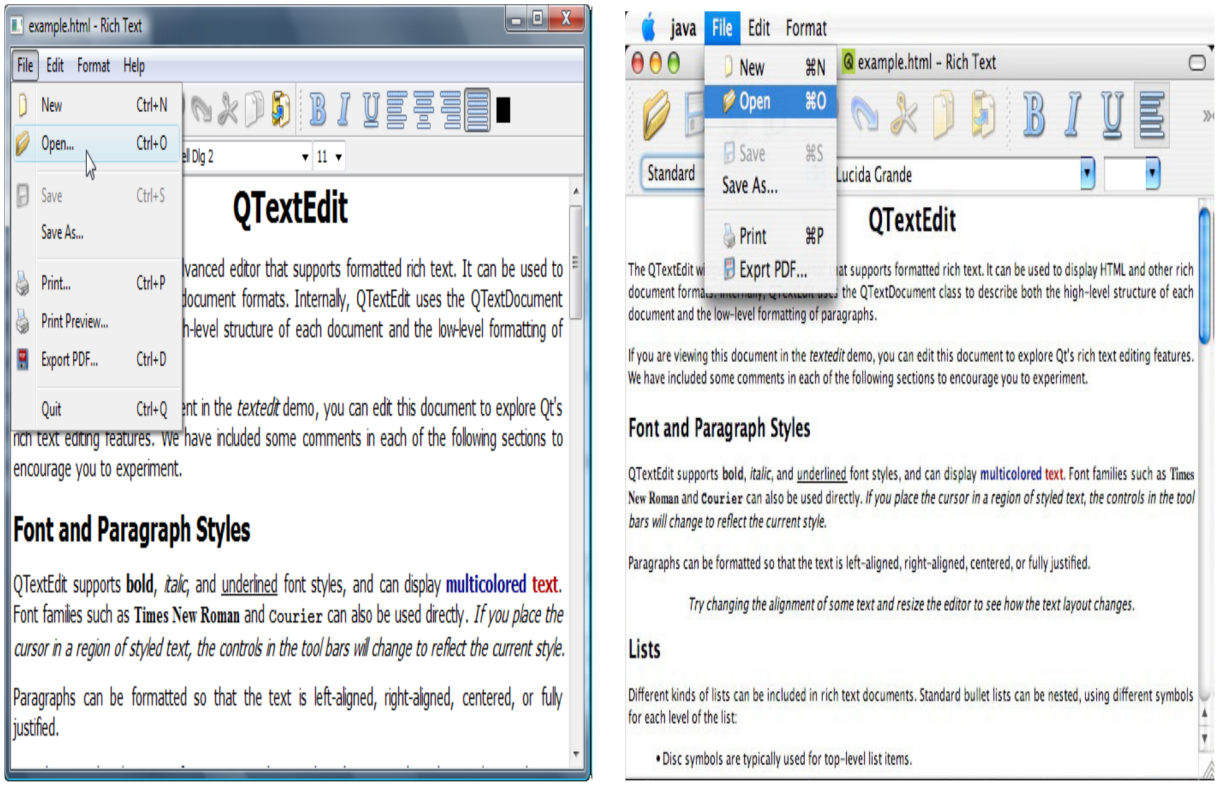

Figure III.04 : Une fenêtre créée avec Qt sous différents OS.

Tout ce qu'on a à faire pour parvenir au même résultat, c'est de recompiler le programme sous chacun des OS [31-32].

III.B.5) Qt Creator :

Qt Creator est l'IDE optimal pour développer des applications avec Qt. Il comprend :

- **Un éditeur de code**, qui permet la coloration syntaxique, la complémentation de code et la recherche d'expressions ;
- **Un éditeur de fenêtres**, qui permet de dessiner le contenu des interfaces ;
- **Une documentation**, qui permet de tout savoir sur Qt [33].

La figure III.05 représente la fenêtre principale de l'IDE Qt Creator :

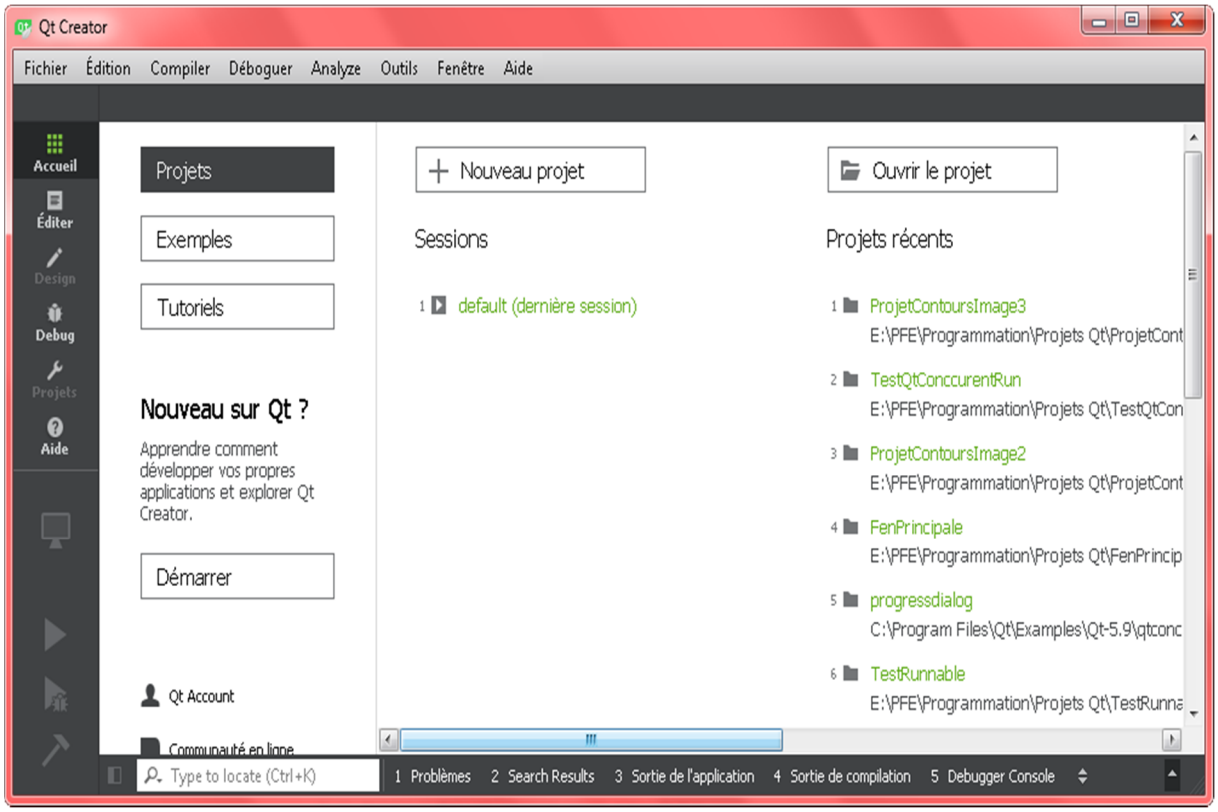

Figure III.05 : Qt Creator.

III.B.6) Qt pour les GUIs :

Application desktop : est une application constituée de fenêtres, contrairement aux applications dites consoles [04].

Pour créer une application desktop avec Qt dans Qt Creator, il suffit tout d'abord de créer un projet Qt, qui contiendra 3 sortes de fichiers :

1- **Un fichier .pro :** Ce fichier, propre à Qt, sert à configurer le projet pour la compilation, il est généré et géré automatiquement par l'IDE ;
2- **Des fichiers .h :** Ce sont les habituels fichiers « *headers* » des projets C++. Il en existe un par fenêtre et contient la déclaration de la sous-classe « FenetreX » héritière de QWidget ;
3- **Des fichiers .cpp :** Ce sont les habituels fichiers sources des projets C++, on en créera un par fenêtre qui contiendra la définition de la sous-classe « FenetreX », plus le fichier « main.cpp » commun à tous les projets Qt, dont le code est donné à la figure III.06 :

```
#include <QApplication>
// Pour accéder à la classe QApplication, qui est la classe de base de
// tout programme Qt.
int main (int argc, char *argv []){
QApplication application (argc, argv); //Création de l'objet application.
Fenetre fen; // Création de la fenetre.
fen.show ();
return application.exec ();} //Démarrage le programme.
```

Figure III.06 : Code de « main.cpp » dans les projets Qt.

Dans les bibliothèques GUIs, dont Qt fait partie, tous les éléments d'une fenêtre sont appelés des widgets, comme : les boutons ou encore les étiquettes. La fenêtre elle-même est considérée être un widget [31].

À présent, nous essayerons de coder la fenêtre représentée en figure III.07 :

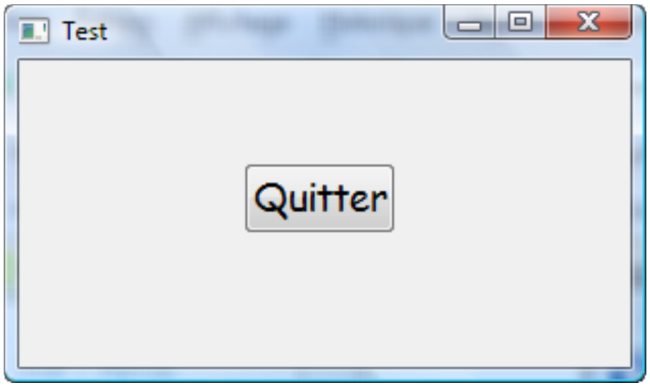

Figure III.07 : La fenêtre « Test » contenant un bouton « Quitter ».

Comme nous avions au préalable donné le code du fichier « main.cpp », Il ne reste à coder que les fichiers « FenetreTest.h » et « FenetreTest.cpp », dont les codes sont énoncés respectivement aux figures III.08 et III.09 :

```
//Fichier FenetreTest.h
#ifndef DEF_FENETRETEST
#define DEF_FENETRETEST
#include <QApplication>
#include <QWidget>
#include <QPushButton>
class FenetreTest : public QWidget // On hérite de QWidget
{
public:
FenetreTest(); // Constructeur
private:
QPushButton *m_bouton ;
};
#endif
```

Figure III.08 : Le code de « FenetreTest.h ».

```
// Fichier FenetreTest.cpp
#include "FenetreTest.h"
FenetreTest ::FenetreTest () : QWidget () //Appel au constructeur de la
// classe mère
{
m_bouton = new QPushButton ("Quitter");// création du bouton Quitter
}
```

Figure III.09 : Le code de « FenetreTest.cpp ».

III.B.6.a) Les widgets:

Il en existe un grand nombre, parmi lesquels nous citerons :

- **Les étiquettes :** de la classe QLabel, permettant d'afficher du texte ou une image sur une fenêtre ;
- **Les champs :** comme ceux de la classe QDoubleSpinBox, permettant de saisir des nombres réels sur une ligne (Voir : figure III.10) ;

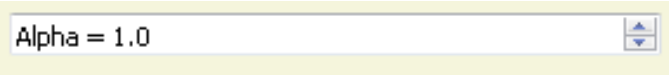

Figure III.10 : QDoubleSpinBox.

- **Les boutons :** comme ceux de la classe QPushButton [31].

La classe QPushButton possède des attributs et des méthodes. Alors, pour modifier les propriétés d'un bouton, il faut tout juste modifier les valeurs de ses attributs, cela étant possible grâce aux accesseurs :

- **var=m_widget->attrib() :** pour récupérer la valeur de l'attribut « attrib » ;
  Exemple : QString texte=m_bouton->text();// pour récupérer le texte du bouton.
- **m_widget->setAttrib(valeur) :** pour modifier la valeur de l'attribut « attrib ».
  Exemple : m_bouton->setToolTip (''Texte d'aide'') ; //pour ajouter une infobulle.

III.B.6.b) Les layouts:

Les classes de layout (QHBoxLayout, QVBoxLayout, QGridLayout et QFormLayout) servent à positionner les widgets de façon relative sur leurs fenêtres.

Pour ce faire, on procède comme suit :

1- Création des widgets ;
2- Création du layout ;
3- Ajout des widgets au layout ;
4- Ajout du layout à la fenêtre [31].

À présent, soit à créer la fenêtre représentée en figure III.11, dans laquelle 3 boutons y sont organisés horizontalement grâce à un layout de la classe QHBoxLayout :

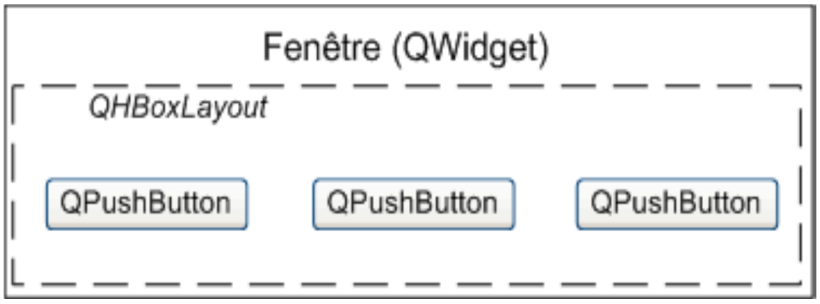

Figure III.11 : Fenêtre à 3 boutons sur layout horizontal.

Le code du fichier « FenetreLayout.cpp » est donné à la figure III.12 :

```
// Fichier FenetreLayout.cpp
#include "FenetreLayout.h"
FenetreLayout::FenetreLayout() : QWidget() //Appel au constructeur de la
// classe mère
{
// Création des boutons
 m_bouton1 = new QPushButton();
 m_bouton2 = new QPushButton();
 m_bouton3 = new QPushButton();
//Création du layout horizontal
 m_hLayout = new QHBoxLayout();
//Ajout des boutons au layout horizontal
 m_hLayout->addWidget(bouton1);
 m_hLayout->addWidget(bouton2);
 m_hLayout->addWidget(bouton3);
//Ajout du layout horizontal à la fenêtre
 this->setLayout(m_hLayout);
}
```

Figure III.12: Le code de la fenêtre organisée selon un layout horizontal.

III.B.6.c) Les signaux et slots:

Les signaux et slots sont des éléments d'un objet qui s'ajoutent aux attributs et méthodes de celui-ci [31].

La figure III.13 montre ce qu'un objet pouvait contenir avant Qt, ainsi que ce qu'il peut contenir maintenant qu'on utilise Qt :

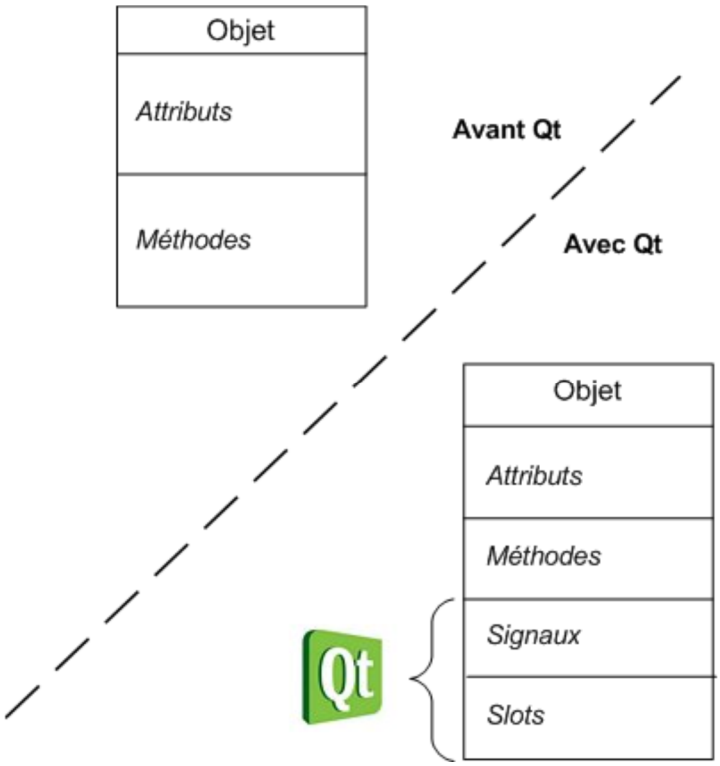

Figure III.13 : Un objet avant et après Qt.

Signal : Est un message émis par un widget lorsqu'un évènement se produit, comme l'évènement « clic sur un bouton » [31].

Slot : Est une méthode connectée à un signal, elle s'exécute à l'émission du signal [31].

Voyons un cas concret (Voir : La fenêtre « Test » contenant un bouton « Quitter » en figure III.07). Nous allons prendre deux objets, d'un côté notre bouton appelé m_bouton (de type QPushButton), et de l'autre notre application (de type QApplication).

Nous voudrions, par exemple, qu'un clic sur le bouton provoque l'arrêt de l'application. Il faudrait, pour cela, connecter le signal « clicked() » (bouton cliqué) au slot « quit() » (quitter l'application). Comme illustré en figure III.14 :

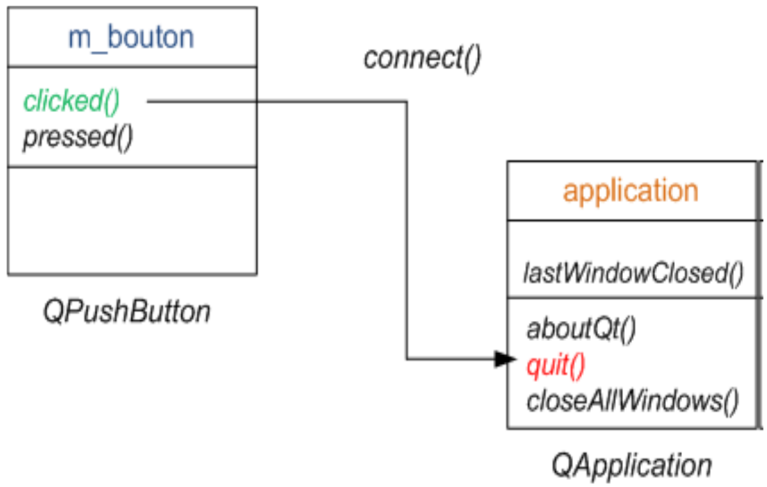

Figure III.14 : Exemple de connexion d'un signal à un slot.

Pour ce faire, il suffit d'ajouter à la fin du constructeur de la classe FenetreTest la ligne de code suivante :

QObject::connect (m_bouton, SIGNAL (clicked ()), qApplication, SLOT (quit ()));

En général les signaux et slots par défaut suffisent, mais il n'est pas rare que l'on veuille en créer de nouveaux, plus appropriés aux besoins. Les signaux ne sont quasiment jamais implémentés par les programmeurs, tant ils en disposent d'une infinité. Contrairement aux slots, qu'on déclare comme suite :

public slots : type_retour slot1 (type_param1 p1, …etc.) ;

Dès que le slot personnalisé est créé, il est nécessaire de définir la macro Q_OBJECT à placer au début de la déclaration de la classe.

III.B.6.d) La fenêtre principale :

Jusqu'à présent, nous avons créé des fenêtres plutôt basiques, en héritant de QWidget. C'est en effet largement suffisant pour de petites applications mais, il est plus judicieux de doter chaque application complexe d'une fenêtre principale.

La classe QMainWindow a été spécialement conçue pour gérer la fenêtre principale de l'application. Parmi les fonctionnalités qu'offre la classe QMainWindow, on trouve notamment la barre des menus, la barre d'outils et la barre d'état [31].

La figure III.15 illustre une fenêtre principale :

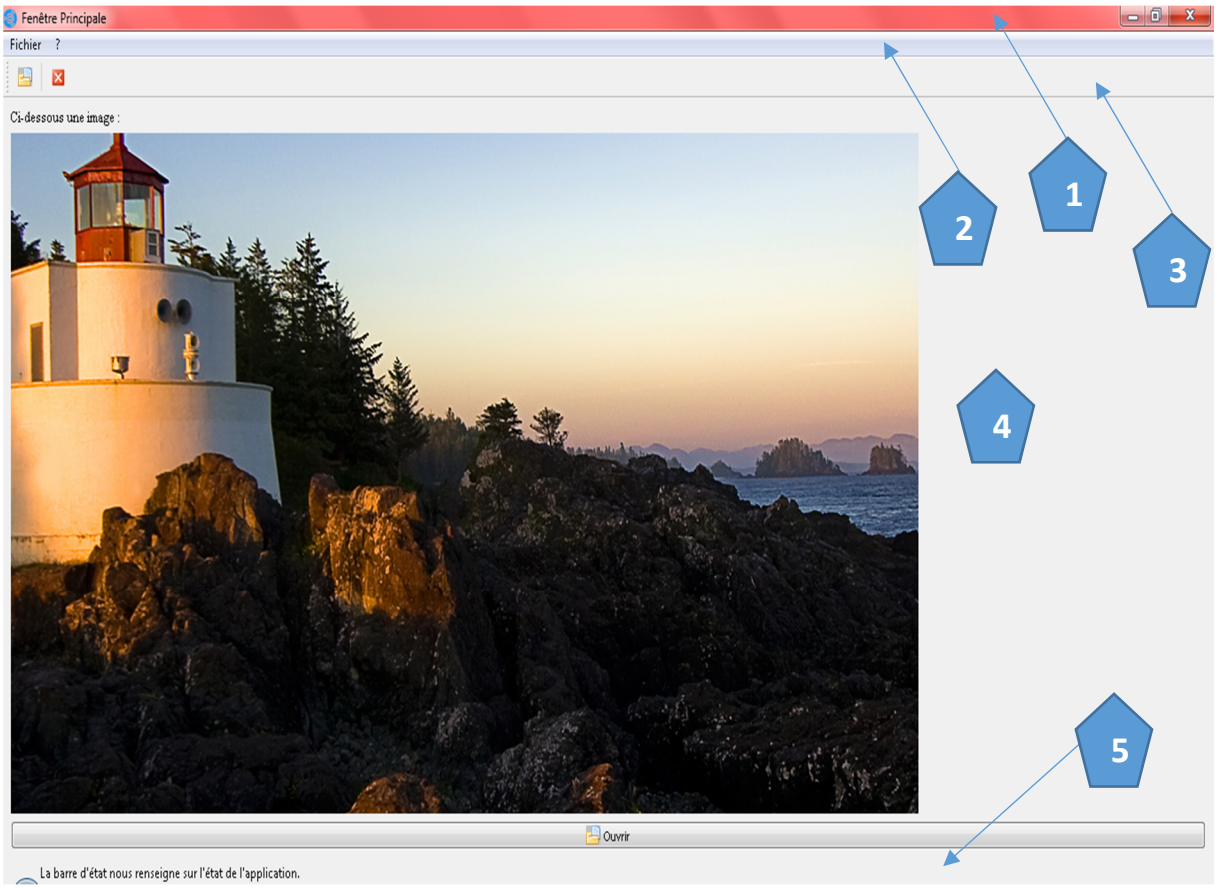

Figure III.15 : Exemple d'une fenêtre principale.

Avec :

1- **Barre de titre :** comporte une icône ainsi qu'un titre ;
2- **Barre des menus :** comporte une suite de menus, lesquels comportent à leurs tour des actions[1] ;
3- **Barre d'outils :** comporte des icônes représentant les actions correspondantes ;
4- **Zone centrale :** il s'agit du cœur de la fenêtre ;
5- **Barre d'état :** renseigne sur l'état de l'application.

[1] Des objets de la classe QAction.

Les figures III.16 et III.17 donnent respectivement les codes des fichiers « FenPrincipale.h » et « FenPrincipale.cpp » de la fenêtre principale en figure III.15 :

```
//FenPrincipale.h
#ifndef FENPRINCIPALE_H
#define FENPRINCIPALE_H
// Les Inclusions
class FenPrincipale : public QMainWindow
{
public:
FenPrincipale(); // Constructeur
private:
//Les Actions
QAction *actionOuvrir;
QAction *actionQuitter;
// Les menus
QMenu *menuFichier;
QMenu *menuPointInterrogation;
//La barre d'outils
QToolBar *barreOutils;
//La zone centrale
QWidget *zoneCentrale;
QVBoxLayout *layoutGlobal;//Layout verticale
QLabel *labelTexte;
QLabel *labelImage;
QPushButton *boutonOuvrir;
//La barre d'état
QStatusBar *barreEtat;
};
#endif // FENPRINCIPALE_H
```

Figure III.16 : Code du fichier «FenPrincipale.h ».

```
// FenPrincipale.cpp
#include "FenPrincipale.h"
FenPrincipale::FenPrincipale():QMainWindow()
{
    //Customisation de la fenetre
    this->setWindowTitle("Fenêtre Principale");//Titre de la fenêtre
    this->setWindowIcon(QIcon("Icones/fenetreIcone.png"));//Icone de la
//fenêtre

    // Les Actions
    //Création des Actions
    actionOuvrir=new QAction("Ouvrir");
    actionQuitter=new QAction("Quitter") ;
    //Customiser les Actions
    //En Ajoutant un raccourci clavier
    actionOuvrir->setShortcut(QKeySequence("Ctrl+O"));
    //En ajoutant une icone
    actionOuvrir->setIcon(QIcon("Icones/ouvrirIcone.png"));
    //En ajoutant une infobulle
    actionOuvrir->setToolTip("Ouvrir un fichier") ;
```

```
    // Les menus
    //Création des menus
    menuFichier=menuBar()->addMenu("&Fichier"); // F souligné en appuyant
//sur ALT
    menuPointInterrogation=menuBar()->addMenu("&?");

    //La barre d'outils
    //Création de la barre d'outils
    barreOutils=addToolBar("Barre D'Outils");

    //Ajout des Actions aux menus
    menuFichier->addAction(actionOuvrir);
    menuFichier->addSeparator();
    menuFichier->addAction(actionQuitter);

    //Ajout des Actions à la barre d'outils
    barreOutils->addAction(actionOuvrir);
    barreOutils->addSeparator();
    barreOutils->addAction(actionQuitter);

    //La zone centrale
    //Création des widgets
    labelTexte=new QLabel("Ci-dessous une image :");
    labelImage=new QLabel()->setPixmap(QPixmap("images/image.png"));
    boutonOuvrir=new QPushButton(QIcon("Icones/ouvrirIcone.png"),"Ouvrir");
    layoutGlobal=new QVBoxLayout();
    zoneCentrale=new QWidget();
    //Customization des widgets
    boutonOuvrir->setToolTip("Ouvrir un fichier");
    labelTexte->setFont(QFont("Times New Roman",10,false));
    //Ajout des widgets au layout global
    layoutGlobal->addWidget(labelTexte);
    layoutGlobal->addWidget(labelImage);
    layoutGlobal->addWidget(boutonOuvrir);
    //Ajout du layout global a la zone centrale
    zoneCentrale->setLayout(layoutGlobal);
    //Ajout de la zone centrale a la fenetre
    setCentralWidget(zoneCentrale);

    //La barre d'état
    barreEtat=new QStatusBar(this);//Création
    this->setStatusBar(barreEtat);// Son ajout à la fenêtre
    barreEtat->showMessage("            La barre d'état nous renseigne sur
l'état de l'application.\n"); // Message à afficher
}
```

Figure III.17 : Code du fichier « FenPrincipale.cpp ».

III.B.7) Qt pour les images :

Les formats d'images supportés, par défaut, par Qt sont :

- .PNG ;
- .BMP ;
- .JPG ;

Qt offre plusieurs classes pour exploiter des images, dont :

- **QImage :** utilisée pour accéder et modifier les pixels des images ;
- **QPixmap :** utilisée pour afficher des images sur les fenêtres [32].

III.B.7.a) La classe QImage :

- **Constructeur :**

Un des constructeurs les plus utilisés de la classe QImage possède le prototype suivant :

QImage::QImage(QString chemin) ;

Avec :

- **chemin :** est soit le chemin absolu du fichier image à partir de la racine du disque, ou bien, le chemin relatif à partir du répertoire où se trouve l'exécutable de l'application, souvent le répertoire « debug ». Il s'agit d'une chaine de caractères (QString) qu'on récupère généralement grâce à l'instruction suivante :

QString chemin = QFileDialog ::getOpenFileName(this, ''Titre de la boite de dialogue'',QString(),''Image(*.png *.bmp *.jpg)'') ;

Avec :

- **QFileDialog :** crée et affiche une boite de dialogue d'ouverture de fichier ;
- **QString() :** Le troisième paramètre de getOpenFileName correspond au nom du répertoire dans lequel l'utilisateur sera positionné à l'ouverture de la boite de dialogue. La valeur QString() équivaut "", donc la boîte de dialogue affiche le répertoire dans lequel est situé l'exécutable ;
- **Image(*.png *.bmp *.jpg) :** n'est qu'un filtre. Seules les images de type PNG, BMP et JPEG s'afficheront dans la boite de dialogue.

L'instruction ci-dessus ouvre une boite de dialogue d'ouverture de fichier (QFileDialog) à partir de laquelle l'utilisateur sélectionne un fichier à ouvrir. La figure III.18 donne un aperçu de cette boite de dialogue :

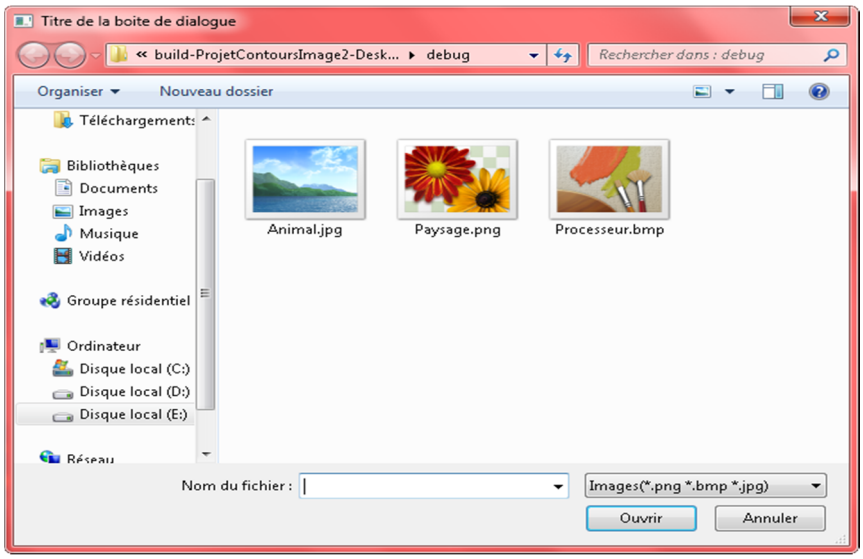

Figure III.18 : Boite de dialogue d'ouverture de fichier.

- **Fonctions membres ou apparentées :**

Le tableau III.1 donne les prototypes et descriptions des fonctions membres ou apparentées à la classe QImage qui s'avèreront très utiles dans notre projet :

Tableau III.1 : Fonctions membres ou apparentées à QImage [32].

	Nom	Prototype	Description
QImage	**width()**	int QImage::width() ;	Retourne la largeur de l'image en nombre de pixels.
	height()	int QImage::height() ;	Retourne la hauteur/longueur de l'image en nombre de pixels.
	pixel(x,y)	QRgb QImage::pixel(int x, int y) ;	Retourne la couleur du pixel de coordonnée (x ; y), à récupérer dans un objet QRgb, qui se présente sous la forme d'un triplet (Red/Rouge ; Green/Vert ; Blue/Bleu) contenant les degrés de présence des couleurs correspondantes dans le pixel. On trouve aussi le setter setPixel(int x, int y, QRgb rgb) ;.
	QGray(rgb)	int QGray(QRgb rgb) ;	Retourne l'intensité de gris d'un triplé rgb en la calculant suivant la formule : (R*11+G*16+B*5)/32

III.B.7.b) La classe QPixmap :

Pour créer un objet de la classe QPixmap, nous procédons de 2 manières distinctes. Soit, l'image n'a pas encore été importée du disque, alors nous ferons appel au constructeur de la classe, dont le prototype est :

QPixmap::QPixmap(QString chemin) ;

Ou bien, nous le construisons à partir d'un objet de la classe QImage, grâce à la fonction fromImage(qImage), dont le prototype est :

QPixmap fromImage(QImage qImage) ;

Une fois que nous avons à notre disposition un objet de la classe QPixmap, il suffira de faire appel à la fonction membre setPixmap(qPixmap) de la classe QLabel afin que l'on puisse afficher l'image sur la fenêtre [32].

III.B.8) Chronomètre de Qt :

Pour chronométrer un traitement, Qt offre la classe QElapsedTimer.

La figure III.19 illustre son utilisation :

```
QElapsedTimer chrono;
chrono.start(); // Démarrage du chronomètre.
// Début du traitement :
TRAITEMENT ();
//Fin du traitement.
qDebug() << "Le traitement a duré : " << chrono.elapsed() << "millisecondes";
```

Figure III.19 : Exemple d'utilisation de QElapsedTimer.

Pour obtenir un résultat plus précis, en nanosecondes au lieu des millisecondes, on remplacera la fonction elapsed() par nsecsElapsed().

QElapsedTimer utilise une horloge monotone, celle-ci est immunisée contre tout changement, induit par l'utilisateur, de l'heure du système ou des paramètres du fuseau horaire.

Dans les systèmes d'exploitation Windows, QElapsedTimer utilise l'une des deux horloges suivantes :

1- **TickCounter :** est monotone, car basée sur le compteur d'impulsions du processeur. Il s'agit néanmoins d'une horloge peu performante ;
2- **PerformanceCounter :** fonctionne sur le même principe que TickCounter mais, est plus performante qu'elle. Pour cela, elle utilise les fonctions Windows QueryPerformanceCounter() et QueryPerformanceFrequency() [36].

La figure III.20 montre le principe d'utilisation des 2 fonctions précédemment citées :

```
#include <Windows.h>
LARGE_INTEGER frequence;
LARGE_INTEGER debut, fin;
double secondesEcoulees;
QueryPerformanceFrequency(&frequence);
QueryPerformanceCounter(&debut);
/* Traitement*/
QueryPerformanceCounter(&fin);
secondesEcoulees = (fin.QuadPart - debut.QuadPart) / (double)frequence.QuadPart;
```

Figure III.20 : Utilisation de QueryPerformanceCounter() et QueryPerformanceFrequency() [37].

Le choix de l'horloge à utiliser par QElapsedTimer se fait automatiquement, sans l'intervention du programmeur. QElapsedTimer préfère l'utilisation de l'horloge PerformanceCounter sauf lorsque celle-ci n'est malheureusement pas supportée par le système [36].

III.B.9) Qt pour faire du parallélisme :

Qt offre de nombreuses classes pour travailler avec des threads. Voici trois approches différentes que les programmeurs peuvent utiliser pour implémenter des applications multithreads :

1- QThread :

API[1] « *Application Programming Interface* » (Interface de programmation d'application) de bas niveau permettant de créer et contrôler des threads ;

2- QThreadPool & QRunnable :

La création et la destruction de threads sont souvent très coûteuses. Pour réduire ces frais, les threads existants peuvent être réutilisés pour de nouvelles tâches.

QThreadPool est une collection de QThread réutilisables.

Chaque application Qt possède « un pool de threads global » accessible via QThreadPool::globalInstance(). Cette collection de threads maintient automatiquement un nombre optimal de threads en fonction du nombre de cœurs du microprocesseur. Pour récupérer ce nombre, on fait appel à la fonction QThreadPool::maxThreadCount().

Le pool de threads global contient au moins un thread, le thread principal, souvent celui de la GUI.

Pour exécuter une tâche dans l'un des threads du « pool de threads global », on doit ré-implémenter la fonction QRunnable::run() dans une sous-classe de QRunnable, puis on instancie cette classe fille. On utilise QThreadPool::start(qRunnable) pour mettre le qRunnable dans la file d'attente du « pool de threads global ». Lorsqu'un des cœurs du microprocesseur devient disponible, le code dans QRunnable::run() s'exécute sur ce cœur ;

3- QtConcurrent :

Le module QtConcurrent fournit des fonctions de haut niveau pour le calcul parallèle. Contrairement à l'utilisation de QThread et QRunnable, ces fonctions ne nécessitent jamais l'utilisation de primitives de synchronisation de bas niveau telles que les mutex ou sémaphores.

[1] Ensemble de fonctions permettant de programmer des applications [04].

Ce module fournit entre autres la fonction QtConcurrent::run(), qui peut exécuter n'importe quelle tâche - enrobée dans une fonction - dans un autre thread que celui de la GUI.

Un objet QFuture peut être utilisé pour récupérer la valeur de retour de ladite fonction et pour recevoir des notifications concernant l'état du thread exécutant celle-ci [36].

Dans notre projet, nous avons choisi de travailler avec QThreadPool & QtConcurrent::run(), pour les raisons suivantes :

- QThreadPool, car :
    - Les frais de création et destruction des threads sont minimisés ;
    - Le pool de thread global maintient un nombre de threads optimal en fonction du nombre de cœurs disponibles sur le microprocesseur.
- QtConcurrent::run(), car :
    - Est très simple d'utilisation, grâce à :
        - Ses fonctions de haut niveau ;
        - Son utilisation ne nécessitant pas le recours aux primitives de synchronisation.

III.B.9.a) QtConcurrent::run() :

- **Exécuter une fonction dans un thread :**

  La figure III.21 donne le code illustrant l'utilisation de QtConcurrent::run() pour exécuter « uneFonction() »dans un thread :

```
Extern void uneFonction() ;
QFuture<void> future = QtConcurrent::run(uneFonction);
```

Figure III.21 : Exécuter une fonction dans un thread.

Si la fonction à exécuter est membre, alors le code à utiliser pour son exécution est donné dans la figure III.22 :

```
// Appeler 'void QImage::invertPixels()'

    QImage image = ...;
    QFuture<void> future=QtConcurrent::run(&image,&QImage::invertPixels);
    Future.waitForFinished(); //Mettre en attente le thread de la GUI
```

Figure III.22 : Exécuter une fonction membre dans thread.

Pour minimiser le coût de création et destruction du thread, nous devrons lancer la fonction « uneFonction() » dans un des threads du pool de threads global, qui ont déjà été créés au lancement de l'application. Le code donné à la figure III.23 montre comment y procéder :

```
extern void uneFonction();
QThreadPool pool = QThreadPool::globalInstance();
QFuture<void> future = QtConcurrent::run(&pool, uneFonction);
```

Figure III.23 : Exécuter une fonction dans un des threads du pool de threads global.

La référence au pool de threads global « &pool »est toujours mise en 1ere position des paramètres de QtConcurrent::run(), devançant ainsi la référence de l'objet « &image », comme indiqué ci-après :

```
QFuture<void> future = QtConcurrent::run(&pool, &image,
                                         &QImage::invertPixels);
```

- **Passer des arguments à la fonction :**

La figure III.24 donne le code permettant de passer des arguments à la fonction :

```
extern void uneFonctionAvecArguments(int arg1, double arg2,QString& arg3);

int entier = ...;
double reel = ...;
QString chaineCar = ...;
QFuture<void> future = QtConcurrent::run(uneFonctionAvecArguments,
entier,reel, chaineCar);
```

Figure III.24 : Passer des arguments à la fonction.

Une copie de chaque argument est faite au moment où QtConcurrent::run() est appelé, puis ces valeurs sont transmises au thread.

- **Récupérer la valeur retournée par la fonction :**

Toute valeur de retour de la fonction est disponible via QFuture. La figure III.25 donne le code permettant de récupérer la valeur retournée par la fonction :

```
extern QString fonctionRetournant1String();

QFuture<QString> future = QtConcurrent::run(fonctionRetournant1String);
QString resultat = future.result();
```

Figure III.25 : Récupérer la valeur retournée par la fonction.

Il faut noter que la fonction QFuture::result() met en attente le thread de la GUI jusqu'à ce que le résultat soit disponible, comme l'avait fait QFuture::waitForFinished() en figure III.22 lorsque le type de retour de la fonction était void.

En résumé, le prototype au complet de QtConcurrent::run() est :

QFuture<Type> QtConcurrent::run(QThreadPool* pool, QClasse* objet, Fonction f, Type_param1 p1, …etc.) ;

Nous remarquons aisément qu'à la suite du lancement du second thread, le thread principale « de la GUI » a souvent tendance à attendre la terminaison de son homologue, soit avec QFuture::waitForFinished(), ou bien avec QFuture::result() le cas échéant, en récupérant le résultat du traitement.

Qu'en serait-il lorsqu'on veuille lancer plusieurs threads simultanément à partir du thread de la GUI, puis les attendre à un point de RDV ?

Pour cela, deux alternatives sont possible, soit :

1) On duplique l'une des deux fonctions bloquantes dans le code ; ou bien, plus élégant encore :
2) On utilise la classe QFutureSynchronizer.

III.B.9.b) QFutureSynchronizer :

QFutureSynchronizer est une classe qui simplifie la synchronisation d'objets QFuture.

Les futures sont ajoutés au synchroniseur à l'aide de la fonction QFutureSynchronizer::addFuture(qFuture). QFutureSynchronizer::waitForFinished() offre un moyen simple pour s'assurer que tous les futures ont achevé leurs travaux, avant de retourner les résultats des fonctions. On utilise QFutureSynchronizer::clearFutures() pour supprimer tous les futures du synchroniseur [36].

La figure III.26 donne un exemple de code où sont utilisées les fonctions citées :

```
QFutureSynchronizer<void> synchroniseur;
synchroniseur.addFuture(QtConcurrent::run(uneFonction));
synchroniseur.addFuture(QtConcurrent::run(uneAutreFonction));
synchroniseur.waitForFinished();
synchroniseur.clearFutures();
```

Figure III.26 : Exemple d'utilisation des fonctions de la classe QFutureSynchronizer.

III.B.9.c) Exemple global de parallélisation avec Qt:

La figure III.27 donne le code d'une application multithread permettant de colorier en rouge les pixels d'une image :

```
#include <QApplication>
#include <QImage>
#include <QElapsedTimer>
#include <QFutureSynchronizer>
#include <QThreadPool>
#include <QtConcurrent/QtConcurrent>
#include <qtconcurrentrun.h>
#include <iostream>

void colorierRouge(int colonneDebut,int colonneFin,QImage* qImage)
{
   int i,j;
for(i=colonneDebut;i<colonneFin;i++)
   for(j=0;j<qImage->height();j++)
   qImage->setPixelColor(j,i,QRgb(255,0,0));
}

int main(int argc, char *argv[])
{
   QApplication app( argc, argv );
   QImage* qImage=new QImage("image.jpg");
   QElapsedTimer* chrono;
   chrono->start();
   QFutureSynchronizer<void>* synchroniseur=new QFutureSynchronizer<void>();
   QThreadPool *pool=QThreadPool::globalInstance();
   int maxThread=pool->maxThreadCount();
   int i;
   for(i=0;i<maxThread;i++)
   synchroniseur->addFuture(QtConcurrent::run(pool,colorierRouge,qImage-
>width()/maxThread*i,qImage->width()/maxThread*(i+1),qImage));
   QThread::yieldCurrentThread();  // Mettre le thread principale en queue de liste du pool
   synchroniseur->waitForFinished();
   std::cout <<"Temps d'execution : "<<chrono->elapsed()<<std::endl;
   synchroniseur->clearFutures();
   qImage->save("imageRouge.jpg");
   return app.exec();
}
```

Figure III.27 : Code d'une application multithread permettant de colorier une image en rouge.

En figure III.28, de droite à gauche, l'image soumise à l'application précédente, suivie de l'image résultante du traitement :

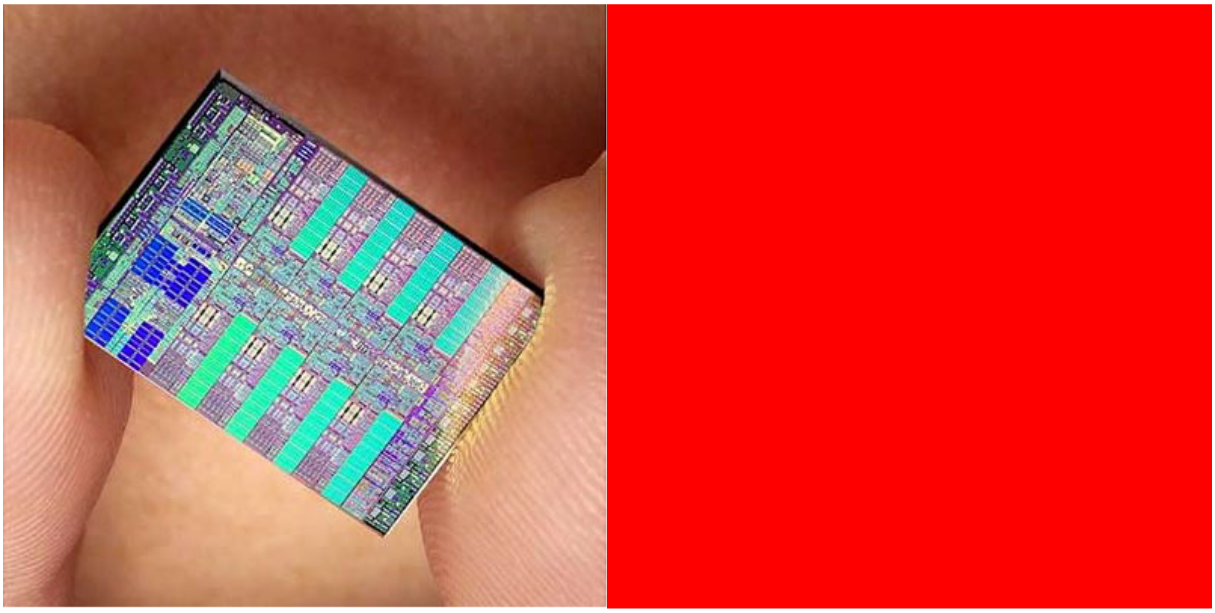

Figure III.28 : L'image soumise à l'application suivie de l'image résultante du traitement.

Ayant effectué une exécution séquentielle puis parallèle du programme précédent sur une machine à microprocesseur bi-cœurs (2 cœurs), une comparaison des exécutions est formulée en figure III.29 :

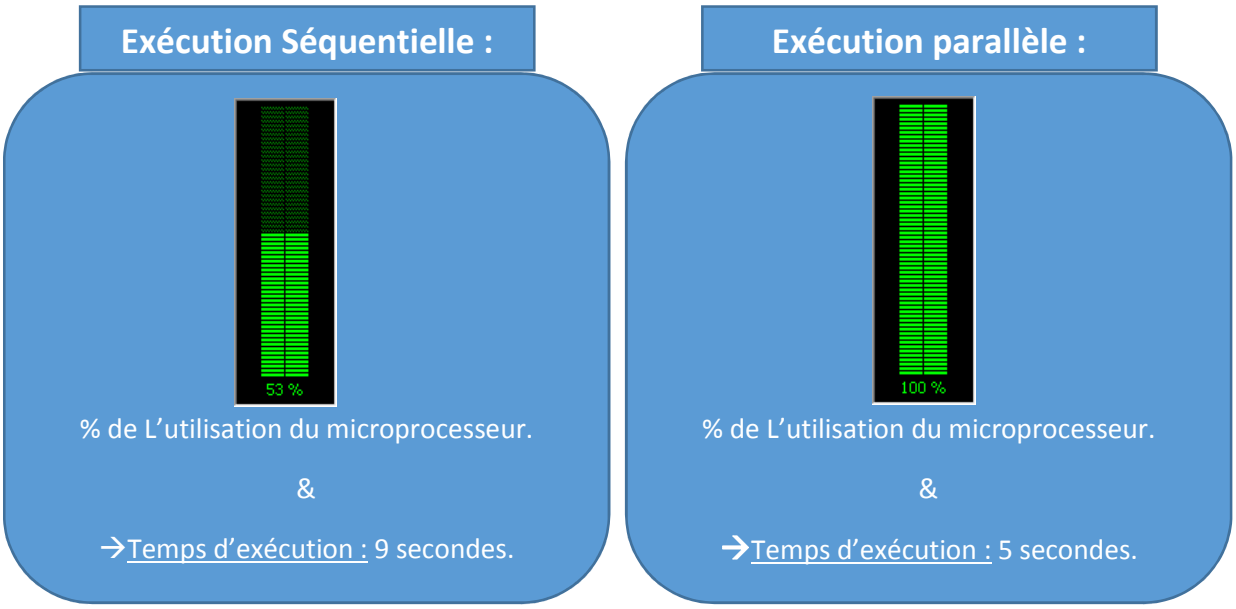

Figure III.29 : Comparaison entre les exécutions séquentielle et parallèle.

Nous remarquons que lors de l'exécution parallèle, le microprocesseur atteint son régime de fonctionnement maximal et le temps d'exécution est pratiquement réduit de moitié, ce qui est dû à l'utilisation des 2 cœurs du microprocesseur en parallèle.

Conclusion :

Au bout de ce chapitre, nous jugeons que les outils qui y sont présentés s'avèrent pertinents, suffisants & de qualité pour la réalisation de notre application, ce qui nous permettra, au chapitre suivant, de nous atteler à la tâche « réalisation de l'application » puis, « expérimentation » de celle-ci.

Chapitre IV : Réalisation & expérimentation.

Ce chapitre est consacré à la réalisation de notre application, qui donne la possibilité d'exécuter en parallèle l'algorithme de R.Deriche pour extraire les contours d'images, puis au déploiement de cette application sur des machines bi/quadri/octocœurs, ainsi que l'expérimentation de celle-ci sur des images de différentes dimensions.

IV.A) Les solutions parallèles retenues :

Sachant qu'il existe 2 types de parallélisme :

1- **Parallélisme de contrôle** ;
2- **Parallélisme de donnée :** utilisé lorsqu'un programme effectue le même traitement sur un grand nombre de données similaires (vecteurs, matrices, …etc.).

Comme l'algorithme de R.Deriche traite des images en niveau de gris, qui ne sont autres que des matrices, dont chaque case contient un entier allant de 0 à 255 signifiant l'intensité de la luminosité du pixel représenté.

Alors, nous choisissons d'adopter le parallélisme de données dans notre application. Cela consiste à diviser l'image source en n blocs de dimensions égales[1], puis appliquer à chacun de ces blocs l'algorithme de R.Deriche.

n est égale au nombre maximum de threads pouvant être lancés simultanément sur une des machines cibles. Il est aussi égal au nombre de cœurs disponibles.

[1] Pour assurer l'équilibrage des charges.

L'algorithme de R.Deriche est implémenté dans la fonction **Deriche()**, dont le prototype est le suivant :

```
void Image::Deriche(Image * image_resultante,int colonne_debut,int colonne_fin);
```

Avec :

- **Image::Deriche :**
  - **Image:: :** La fonction **Deriche()** est appelée par l'objet image_source « L'image à traiter » de type **Image**, comme suit :

    Image_source->Deriche(..) ;
  - **Deriche()** : Est une fonction membre de la classe **Image**. Cette classe simplifie et optimise la classe de Qt **QImage**.
- **colonne_debut/fin** : Délimitent, par un découpage vertical, la portion de l'image source à traiter.

La figure IV.1 représente un schéma décrivant le parallélisme de donnée de l'algorithme de R.Deriche sur n threads :

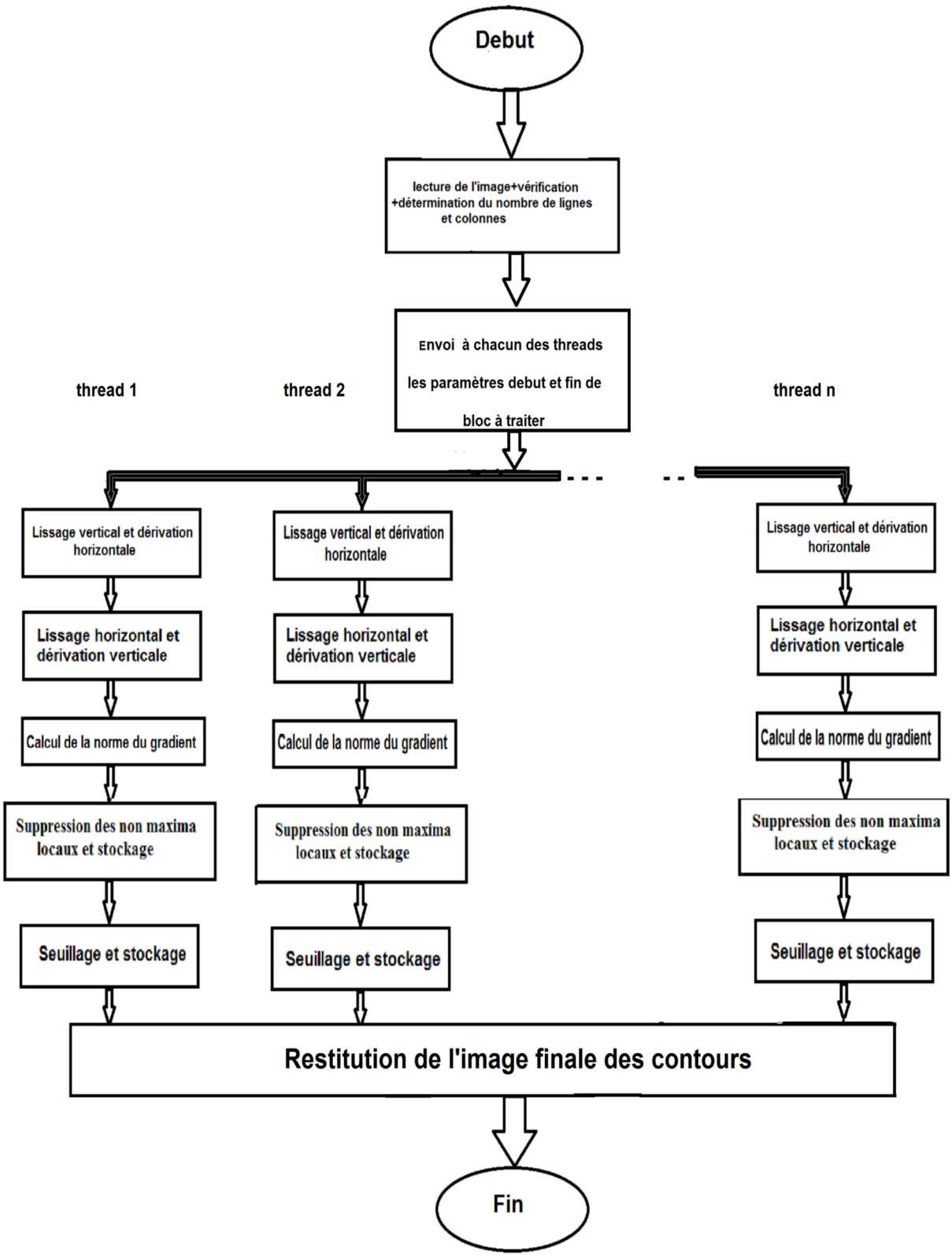

Figure IV.1 : Parallélisme de donnée de l'algorithme de R. Deriche sur n threads.

Le parallélisme de donnée offre 2 possibles implémentations :

1- Décomposition par blocs ;
2- Décomposition cyclique.

IV.A.1) Décomposition par blocs :

L'implémentation de la décomposition par blocs est donnée à la figure IV.2 :

```
    QThreadPool *threadPool =QThreadPool::globalInstance();
    int maxThread =threadPool->maxThreadCount();
    QFutureSynchronizer<void> *synchroniser =new QFutureSynchronizer<void>()
    for(int i=0;i<maxThread;i++)
    {
     Synchronizer->addFuture(QtConcurrent::run(threadPool, image_source,
&Image::Deriche, image_resultante, i*(image_source->width)/maxThread,
(i+1)*(image_source->width)/maxThread));
    }
    QThread::yieldCurrentThread();
    synchroniser->waitForFinished();
    synchroniser->clearFutures();
```

Figure IV.2 : Implémentation de la décomposition par blocs.

IV.A.2) Décomposition cyclique :

L'implémentation de la décomposition cyclique est quasi identique à celle par blocs, mais au lieu de lancer directement la fonction Deriche(), nous faisons d'abord appel à la fonction DericheCyc(), dont le code est donné à la figure VI.3 :

```
void Image::DericheCyc(Image * dst,int largeur_unit,int numThread){
    int debut,fin,i=0;
    debut=numThread*largeur_unit;
    while(i<this->cycles)
    {
       fin=debut+largeur_unit;
       this->Deriche(dst,debut,fin);
       i++;
       debut=debut+(this->width/this->cycles);
    }
}
```

Figure IV.3 : Code de la fonction DericheCyc().

Avec :

- **`largeur_unit`** : fixée à `image_source->width/(image_source->cycles*maxThread)`. Elle représente la dimension d'un bloc de l'image.
- **`this->cycles`** : est le nombre de cycles du traitement, est fixé à **`2`**. Autrement dit, est le nombre de blocs que traitera chaque thread.
- **`numThread`** : est le numéro du thread, qui commence à zéro (0).

IV.B) Présentation de l'application :

Nous avons réalisé notre application en utilisant les outils suivants :

- Le langage C++ 11, dont dernière modification en septembre 2011;
- Le « *framework* » Qt 5.9.0, dont dernière modification le 30/05/2017 à 11H59 (UTC) ;
- L'IDE Qt Creator 4.2.1, dont dernière modification le 13/12/2016 à 14H40 (UTC).

La fenêtre d'accueil de notre application est représentée en figure IV.4 :

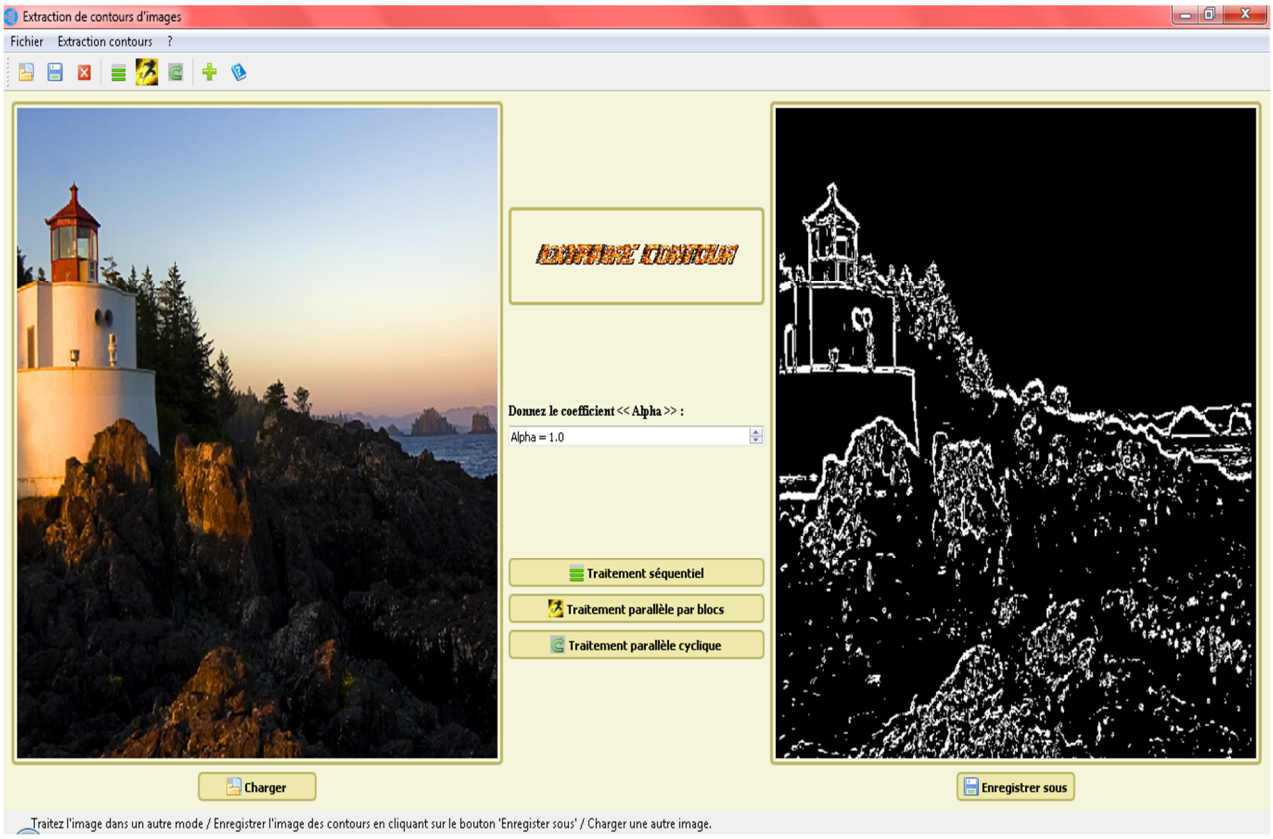

Figure IV.4 : Fenêtre d'accueil de notre application.

Cette fenêtre est principale, de type QMainWindow, elle contient de haut en bas :

- **Une barre de titre** ;
- **Une barre des menus** : contenant les menus suivants :
  - **Fichier** : contenant les actions : Charger, Enregistrer sous & Quitter ;
  - **Extraction contours** : contenant les actions suivantes : Traitement séquentiel, parallèle par blocs/cyclique ;
  - **?** : contenant les actions suivantes : À propos & Aide.
- **Une barre d'outils** : contenant toutes les actions de la barre des menus ;
- **Une zone centrale** : contenant :
  - **Un bouton « Charger »** : pour sélectionner à partir du disque une image à traiter ;
  - **Un QDoubleSpinBox** : pour déterminer la valeur du paramètre « Alpha », avant de passer au(x) traitement(s) ;
  - **Des boutons « Traitement »** : pour exécuter l'algorithme de R.Deriche sur l'image source chargée, selon l'un des traitements possibles (séquentiel, parallèle par blocs/cyclique) ;
  - **Un bouton « Enregistrer sous »** : pour sauvegarder l'image résultante du traitement sur disque ;
  - **Des Étiquettes** : de type QLabel pour afficher respectivement de gauche à droite l'image source chargée et l'image résultante du traitement.
- **Une barre d'état** : qui renseigne sur les actions exécutables à chaque étape d'utilisation de l'application.

IV.C) Expérimentation :

À présent, nous passons à l'expérimentation de notre application. Nous commençons par présenter les caractéristiques des différentes machines, puis celles des images utilisées à cet effet.

IV.C.1) Les machines utilisées :

Le tableau IV.1 décrit les caractéristiques des machines utilisées pour l'expérimentation :

Tableau IV.1 : Caractéristiques des machines utilisées pour l'expérimentation.

Machine	Nom	Processeur	Nombre Cœurs	Fréquence Cœurs (GHz)	Plateforme
1) Bicœur :	Toshiba Satellite C55-B1336	Intel Celeron N2820	2	2.13	Microsoft Windows 7 Professionnel Service pack 1
2) Quadricœur :	Dell OptiPlex 790	Intel Core i5 2400	4	3.10	Microsoft Windows 7 Professionnel Service pack 1
3) Octocœur :	Toshiba Satellite L50A1DG	Intel Core i7 4700MQ	8	2.70	Microsoft Windows 8.1

Nous remarquons que :

- Les fréquences d'horloge des microprocesseurs présentés ne dépassent pas les 4 GHz ;
- Les machines présentent des architectures différentes, ce qui constitue une multitude de paramètres pouvant influer sur l'expérimentation et provoquer de légères variations, comme :
  - La technologie TurboBoost qui permet qu'aux microprocesseurs i5 et i7 d'augmenter la fréquence d'horloge des cœurs actifs lorsque tous ne le sont pas.
- Les différentes plateformes ne poseront pas de problème, car notre application est grâce à Qt multiplateforme.

IV.C.2) Les images utilisées :

Les images à soumettre à notre application ont différentes dimensions allant de 400*400 à 2000*2000 et représentent une même photographie satellitaire, prise le 17 mars 2014, par le télescope spatial Hubble de la nébuleuse[1] du singe[2] NGC 2174 située dans la constellation d'Orion à environ 6400 années-lumière de la terre. Cette photographie est donnée en figure IV.5 :

Figure IV.5 : Photographie de la nébuleuse du singe NGC 2174 [38].

[1] Désigne, en astronomie, un objet céleste composé de gaz raréfié, ionisé qui absorbe la lumière d'une étoile proche et la réémet sous forme de couleurs variées [39].
[2] Est appelée ainsi, car elle ressemble à la tête d'un singe [39].

Les contours de la nébuleuse NGC 2174 extraits par notre application sont représentés en figure IV.6 :

Figure IV.6 : Contours de la nébuleuse du singe NGC 2174.

Le tableau IV.2 décrit les caractéristiques des images utilisées pour l'expérimentation :

Tableau IV.2 : Caractéristiques des images utilisées pour l'expérimentation.

Image	Dimension(Pixel²)	Taille (KO)
1) Petite :	400*400	197
2) Large :	1200*1200	1720
3) Entière :	2000*2000	2939

Nous remarquons que l'image 2 (respectivement 3) possède 9 (respectivement 25) fois plus de pixels que l'image 1.

Il est aussi à souligner que l'extension des images importe peu et puis que notre application permet le traitement de toutes les extensions. Et une fois que l'image résultante du traitement est prête, elle sera enregistrée sous l'extension « .png », car ainsi elle occupera moins d'espace disque sans pour autant que la qualité de l'image n'en soit dégradée[1].

IV.C.3) Évaluation du traitement séquentiel :

Dans cette partie, nous évaluerons l'exécution séquentielle de l'algorithme de R.Deriche, en faisant varier, à tour de rôle, les 2 paramètres suivants :

1- La fréquence d'horloge ;
2- La dimension des images ;

IV.C.3.a) La fréquence d'horloge :

Le tableau IV.3 donne les temps du traitement séquentiel de l'image 2 sur les différentes machines :

Tableau IV.3 : Temps du traitement séquentiel de l'image 2 sur les différentes machines.

Machine	Fréquence d'horloge (GHz)	Temps d'exécution (Seconde)
1) Bicoeurs :	2.13	3.037
2) Quadricoeur :	3.10	1.037
3) Octocoeur :	2.70	1.385

[1] Le format png utilise une compression d'images sans perte de données, ce qui permet de réduire la taille de l'image sur le disque tout en sauvegardant sa qualité.

Nous remarquons que :

- Le temps d'exécution n'est pas affecté par le nombre de cœurs des microprocesseurs ;
- Le temps d'exécution est inversement proportionnel à la fréquence d'horloge des cœurs. Cette proportion n'est pas linéaire, et cela étant dû à l'activation de la technologie TurboBoost dans les machines quadri & octocœurs.

IV.C.3.b) La dimension des images :

Le tableau IV.4 donne les temps du traitement séquentiel des images 1, 2 et 3 sur la machine Bicœur :

Tableau IV.4 : Temps du traitement séquentiel des images 1, 2 et 3 sur la machine Bicoeur.

Image	Dimension (Pixel^2)	Temps d'exécution (Seconde)
1) Petite :	400*400	0.329
2) Large :	1200*1200	3.029
3) Entière :	2000*2000	9.331

Nous remarquons que :

- Le temps d'exécution est proportionnel à la dimension des images.
- Le temps du traitement de l'image 2 (respectivement 3) est plus de 10 (respectivement 29) fois plus conséquent à celui de l'image 1. Or, nous nous attendons à ce qu'il soit 9 (respectivement 25) fois plus conséquent. Cela pourrait s'expliquer soit par la grande taille des images 2 et 3 qui engendre de la latence mémoire, ou alors, la lenteur d'exécution qui ouvre la voie à des processus démons demandeurs de la ressource « cœur », impliquant des permutations de contexte, dont le coût temporel s'ajoute au temps d'exécution chronométrée [40].

IV.C.4) Évaluation du traitement parallèle :

Dans cette partie, nous passons à l'évaluation des exécutions parallèles de l'algorithme de R.Deriche pour l'extraction de contours des 3 images présentées précédemment sur les 3 machines utilisées à cet effet.

Le tableau IV.5 donne (en seconde) les temps des traitements parallèles des images 1, 2 & 3 sur les différentes machines :

Tableau IV.5 : Temps (en seconde) des traitements parallèles des images 1, 2 & 3 sur les différentes machines.

	Bicœur			Quadricœur			Octocœur		
	Séquentiel	Parallèle par blocs	Parallèle cyclique	Séquentiel	Parallèle par blocs	Parallèle cyclique	Séquentiel	Parallèle par blocs	Parallèle cyclique
1) Petite (400*400)	0.329	0.174	0.189	0.150	0.049	0.051	0.150	0.038	0.048
2) Large (1200*1200)	3.037	1.565	1.587	1.037	0.298	0.319	1.385	0.294	0.317
3) Entière (2000*2000)	9.331	4.678	4.819	2.983	0.795	0.819	3.945	0.766	0.812

Nous remarquons que :

- Les traitements parallèles sont moins lents que les traitements séquentiels.
- Les traitements parallèles cycliques sont plus lents que ceux par blocs.
- Plus il y a de cœurs, mieux se portent les traitements parallèles.

Dans la suite, nous introduisons certains critères d'évaluation de programme parallèle.

IV.C.4.a) Accélération :

L'accélération, donnée sans dimension, notée *Acc*, est le rapport entre le temps d'exécution séquentielle sur le temps d'exécution parallèle d'un même programme sur une même machine. Elle est donc calculée comme suit :

$$Acc = Ts / Tp \qquad (1)$$

Avec :

- **Ts :** Temps d'exécution séquentielle.
- **Tp :** Temps d'exécution parallèle.

Sa borne supérieure égale le nombre de cœurs *N* du microprocesseur (Acc ≤ *N*) [41].

Le tableau IV.6 donne les accélérations des traitements parallèles effectués précédemment :

Tableau IV.6 : Accélérations des traitements parallèles.

	Bicoeur		Quadricoeur		Octocoeur	
	Par blocs	Cyclique	Par blocs	Cyclique	Par blocs	Cyclique
1) Petite :	1.89	1.74	3.06	2.94	3.94	3.13
2) Large :	1.94	1.91	3.48	3.25	4.71	4.37
3) Entière :	1.99	1.94	3.75	3.64	5.15	4.85

Nous remarquons que l'accélération tend vers 2 (respectivement 4) sur la machine bicœur (respectivement quadricœur). Et nous avons espéré qu'elle tende vers 8 sur la machine octocœur, or cela n'est pas le cas, car cette dernière possède 8 cœurs logiques et seulement 4 de physiques[1].

Nous remarquons que l'accélération est proportionnelle à la dimension de l'image. Pour les petites images, l'accélération est moins convaincante, car le coût de la mise en place du parallélisme est important par rapport au temps du traitement effectif.

IV.C.4.b) Efficacité :

L'efficacité, donnée sans dimension, notée *Eff*, traduit le taux d'utilisation des *N* cœurs de la machine, désignée par le rapport entre l'accélération *Acc* et le nombre *N* des cœurs. Elle est donc calculée comme suit :

$$Eff = Acc / N \qquad (2)$$

L'efficacité est un taux, donc limitée par 1 [41].

[1]Ce type de machines jouit de la technologie « *hyper-threading* » d'Intel, qui consiste à créer deux cœurs logiques sur un seul physique, chacun doté de ses propres registres, et d'un contrôleur d'interruptions particulier. Ces deux unités partagent le cache et le bus système. Ainsi, deux threads peuvent être traités simultanément par le même cœur physique, par contre les performances enregistrées seront nettement inférieures à celles qu'on obtient sur deux cœurs physiques [42].

Le tableau IV.7 donne les efficacités des traitements parallèles effectués précédemment:

Tableau IV.7 : Efficacités des traitements parallèles.

	Bicoeur		Quadricoeur		Octocoeur	
	Par blocs	Cyclique	Par blocs	Cyclique	Par blocs	Cyclique
1) Petite :	0.94	0.87	0.76	0.74	0.49	0.39
2) Large :	0.97	0.95	0.87	0.81	0.59	0.55
3) Entière :	0.99	0.97	0.94	0.91	0.64	0.61

Nous remarquons que ces taux sont assez élevés, ce qui nous permet d'exprimer notre entière satisfaction des outils utilisés pour la réalisation de notre application.

Le tableau IV.8 effectue une comparaison entre la valeur moyenne des efficacités enregistrées sur notre application et celles d'autres applications, similaires à la nôtre, réalisées à l'aide d'outils de parallélisations autres que Qt :

Tableau IV.8 : Comparaison des efficacités des outils de parallélisations.

	ParaTask-Java	Intel TBB	Qt
Efficacité moyenne	0.66 [43]	0.49 [02]	0.78

Nous remarquons que l'outil de parallélisations Qt, nous a permis de réaliser une application nettement plus efficace que celles développées avec ParaTask-Java ou Intel TBB. Surtout que, les développeurs de ces applications « similaires » les ont intentionnellement déduit d'interfaces graphiques ergonomiques au profit de leurs performances, car ainsi le thread global serait moins surchargé. À contrario, nous avions adopté un interfaçage assez ergonome pour que l'expérience utilisateur ne s'en trouve pas désagréable.

Nous ajouterons que ces applications « similaires » n'ont été testées que sur des machines bicœurs. Or, Qt sur ce type de machine réalise plutôt une efficacité moyenne de 0.95.

Pour finir, il est aussi à noter que ces applications « similaires » n'implémentent que le parallélisme par blocs. Or, Qt sur ce type de parallélisme atteint une efficacité moyenne de 0.97.

Conclusion :

Ce chapitre démontre que nous avons atteint l'objectif du mémoire, qui n'est autre que la réalisation d'une application « *desktop* » permettant l'exécution séquentielle & parallèle de l'algorithme de R.Deriche pour l'extraction de contours d'images en utilisant l'outil Qt, lequel s'est avéré très performant dans cet exercice.

CONCLUSION GÉNÉRALE & PERSPECTIVES

Conclusion générale & Perspectives :

Les microprocesseurs ont connu une évolution sans précédent avec l'arrivée des architectures multicœurs. Ce type d'architectures permet de maintenir l'augmentation des performances connue jusqu'alors, tout en apportant une solution aux difficultés de dissipation thermique des microprocesseurs. Pour garder des performances compétitives, les applications doivent donc désormais tirer parti du parallélisme exposé par les architectures multicœurs.

Au cours de la réalisation de notre mémoire, nous avons implémenté l'algorithme de R.Deriche servant à l'extraction de contours d'images, en exploitant le parallélisme sur des microprocesseurs multicœurs grâce au « *framework* » Qt. Ce qui nous a permis d'obtenir des exécutions à efficacité moyenne de près de 80 %. Ces résultats s'ajoutent à l'ensemble des études menées sur le même thème. Ainsi, nous avons constaté que Qt facilite l'implémentation du parallélisme d'une part, et tire au mieux parti des ressources de calcul des machines cibles allant jusqu'à effleurer les bornes supérieures des scalaires d'évaluation d'autre part, tout en préservant une expérience ergonome à l'utilisateur, et cela bien mieux que d'autres outils cités au chapitre IV.

Il est néanmoins à déplorer sur l'aspect méthodologique & empirique de notre étude expérimentale l'utilisation de machines à architectures non équivalentes qui entraine des observations plus au moins explicables. S'ajoute à cela le compromis entre efficacité & ergonomie auquel nous avions contraint notre application.

Des perspectives dans le domaine sont envisageables. Nous pouvons citer l'avènement des MIC « *Many Integrated Cores* », des microprocesseurs constitués de plusieurs dizaines de cœurs, comme l'Intel Xeon Phi 7100 en 2013 avec 61 cœurs. Ceci fera sans nul doute appel à de nouveaux modèles de programmation parallèle plus amène à tirer profit de telles architectures, tels que : Cilk, Grand Central Dispatch, OpenMP, Microsoft's Task Parallel Library. Ou encore l'émergence de ce que nous appelons le « *Semantique Computing* », dont l'idée principale est d'approvisionner l'environnement d'exécution en un lot suffisant d'informations sémantiques afin qu'il décide seul & de façon autonome de la meilleure manière d'exécuter les programmes [15].

Liste des références bibliographiques :

[01] Dumartin, T. (2004). Architecture des ordinateurs : notes de cours. Repéré à http://www.geea.org/IMG/pdf/Cours_II.pdf

[02] Bouzeboudja, A. Hassani, M. (2016). Traitement parallèle pour l'extraction d'image dans un environnement multi cœurs [Mémoire master]. Université Mouloud Mammeri de Tizi-Ouzou.

[03] Architecture de Von Neumann. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 13 mai 2017 à https://fr.wikipedia.org/wiki/Architecture_de_von_Neumann

[04] Cherti, A. (2006). Jargon Informatique (version 1.3.6) [Logiciel].

[05] Djema, L. (2016). Architecture II : notes du cours 1. Recueil inédit. Université Mouloud Mammeri de Tizi-Ouzou.

[06] Djema, L. (2016). Architecture II : notes du cours 4. Recueil inédit. Université Mouloud Mammeri de Tizi-Ouzou.

[07] Anthony, L. (2014). Processeur. Repéré à https://www.vulgarisation-informatique.com/processeur.php

[08] Signal d'horloge. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 13 mai 2017 à https://fr.wikipedia.org/wiki/Signal_d%27horloge

[09] Fréquence d'horloge. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 13 mai 2017 à https://fr.wikipedia.org/wiki/Fr%C3%A9quence_d%27horloge

[10] Geneves, S. (2013). Étude de performances sur processeurs multi cœur : environnement d'exécution évènementiel efficace et étude comparative de modèles de programmation (Thèse doctorat, Université de Grenoble). Repéré à https://tel.archives-ouvertes.fr/tel-00842012/document

[11] Méthodes de refroidissement pour ordinateurs. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 13 mai 2017 à https://fr.wikipedia.org/wiki/M%C3%A9thodes_de_refroidissement_pour_ordinateur

[12] Guillemin, C. (2005). Pourquoi Intel et AMD se convertissent au puces double cœur. Repéré à http://www.zdnet.fr/actualites/pourquoi-intel-et-amd-se-convertissent-aux-puces-double-coeur-39228790.htm

[13] Microprocesseur multi-cœur. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 09 mai 2017 à https://fr.wikipedia.org/wiki/Microprocesseur_multi-c%C5%93ur

[14] Lecrabe. (2016). Le rôle des processeurs et leurs cœurs. Repéré à https://lecrabeinfo.net/le-role-des-processeurs-et-de-leurs-coeurs.html

[15] Vajda, A. (2011). Programming Many-Core Chips. New York, USA: Springer science & business media.

[16] Prise en charge des processeurs multicœurs dans AutoCAD. (s. d.). Repéré à https://knowledge.autodesk.com/fr/support/autocad/learn-explore/caas/sfdcarticles/sfdcarticles/FRA/Support-for-multi-core-processors-with-AutoCAD.html

[17] Parallélisme (informatique). (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 09 mai 2017 à https://fr.wikipedia.org/wiki/Parall%C3%A9lisme_(informatique)

[18] Turcotte, K. Guillemette, A. (2007). Projet de recherche Cégep de Sainte-Foy sur le multithreading. Repéré à http://www2.cegep-ste-foy.qc.ca/departements/freesite-informatique/ProjetsRechercheH2007/H07-620-Equipe05/theorie.html

[19] Chopard, B. (1995). Parallélisme de données et de contrôle. Repéré à http://cuiwww.unige.ch/~chopard/ATO-II/cours/node10.html

[20] Blaise, B. (2017) Introduction to Parallel Computing. Repéré à https://computing.llnl.gov/tutorials/parallel_comp/

[21] Goubault, E. (2009). Écriture de programmes parallèles. Repéré à http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/poly/cours008.html

[22] Lingrand, D. (2008). Introduction au traitement d'images. Paris, France : Vuibert.

[23] Djema, L. Taguemount, A. Boumghar, FO. (2005, Mars). Traitement Parallèle sous MPI-2, pour l'Accélération de l'Algorithme d'Extraction de Contours d'Images. Communication présentée au 3 rd International Conference: Sciences of Electronic, Technologies of Information and Telecommunications, Tunis, Tunisie. Repéré à http://www.setit.rnu.tn/last_edition/setit2005/applications/387.pdf

[24] Vialar, A. (S. D.). IUT Informatique de Bordeaux / Licence Pro DAWIN Traitement d'Image. Repéré à http://dept-info.labri.fr/~vialard/Traitement/

[25] Deriche edge detector. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 28 mars 2017 à https://en.wikipedia.org/wiki/Deriche_edge_detector

[26] Nebra, M. Schaller, M. (2011). Programmer avec le langage C++. Paris, France: OpenClassrooms.

[27] Stoustrup, B. (2013). The C++ Programming Language. New Jersey, USA: Pearson Education.

[28] C++. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 15 juillet 2017 à https://fr.wikipedia.org/wiki/C%2B%2B

[29] Lazarte, M. (2011). Le langage C++ remarqué pour sa performance grâce à une nouvelle norme ISO/CEI. Repéré à https://www.iso.org/fr/news/2011/10/Ref1472.html

[30] Bersini, H. (2009). La programmation orientée objet. Paris, France : Eyrolles.

[31] Molkentin, D. (2007). The Book of Qt 4: The Art of Building Qt Applications. San Fransisco, USA: No Starch Press.

[32] Thelin, J. (2007). Foundations of Qt development. New York, USA: Apress.

[33] Benabadji, A. (2014). Réalisation d'une application de gestion dans un centre d'imagerie médicale (Mémoire master, Université Abou Bekr Belkaid de Tlemcen). Repéré à http://dspace.univ-tlemcen.dz/bitstream/112/6001/1/Realisation-dune-application-de-gestion-dans-un-centre-dimagerie-medicale.pdf

[34] Qt. (s. d.). Dans Wikipédia, l'encyclopédie libre. Repéré le 15 juillet 2017 à https://fr.wikipedia.org/wiki/Qt

[35] The Qt Company. (2017). Licensing. Repéré à https://www.qt.io/licensing/

[36] The Qt Company. (2017). Qt Assistant (version 5.9.0) [Logiciel].

[37] David, A. (2015). Monotonic Clocks – the Right Way to Determine Elapsed Time. Repéré à https://www.softwariness.com/articles/monotonic-clocks-windows-and-posix/

[38] NASA. (2014). NGC2174 [image.png]. Repéré à http://hubblesite.org/image/3336/gallery

[39] Demeersman, X. (2014). Hubble revisite la nébuleuse de la tête de singe. Repéré à http://www.futura-sciences.com/sciences/actualites/astronautique-hubble-revisite-nebuleuse-tete-singe-52862/

[40] Perrot, K. (2014). Organisation, Processus et Interruptions. Repéré à http://pageperso.lif.univ-mrs.fr/~kevin.perrot/documents/2014/systeme/Cours01_14.pdf

[41] Gouarin, L. Louvet, V. Series, L. (2012).Mesure de l'accélération et efficacité du parallélisme [Présentation Powerpoint]. Repéré à http://calcul.math.cnrs.fr/Documents/Ecoles/LEM2I/Mod3/paral.pdf

[42] Casey, SD. (2011). How to Determine the Effectiveness of Hyper-Threading Technology with an Application. Repéré à https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application/

[43] Khaled, B. (2011). Parallélisme dans les applications desktop [Mémoire master]. Université Mouloud Mammeri de Tizi-Ouzou.

Résumé

Afin d'augmenter au maximum l'utilisation des ressources microprocesseurs dans les architectures multicœurs, l'implémentation de programmes parallèles est un besoin qui s'impose.

Ce mémoire aborde l'utilisation du « *framework* » Qt pour la parallélisation de l'algorithme de R.Deriche servant à l'extraction de contours d'images.

Afin de mesurer la pertinence de l'outil Qt dans cet exercice, une expérimentation suivie d'une évaluation comparative furent réalisées. Les résultats de celles-ci donnèrent que Qt facilite l'implémentation du parallélisme, tire au mieux parti des ressources des multicœurs, tout en préservant une expérience ergonome à l'utilisateur, et cela bien mieux que d'autres outils cités. Ce qui fait de Qt une référence dans la réalisation de programmes parallèles.

Mots clés : multicœur, parallélisme, Qt, Deriche.

Abstract

In order to maximize the use of microprocessor resources in multicore architectures, the implementation of parallel programs is a necessity.

This paper deals with the use of the Qt framework for the parallelization of Deriche's algorithm used to detect image edges.

In order to measure the relevance of the Qt tool in this exercise, an experiment followed by a comparative evaluation were carried out. The results of these tests made it possible for Qt to facilitate the implementation of parallelism, to take full advantage of the multicore resources, while preserving an ergonomic experience for the user, much better than other tools cited. This makes Qt a reference in the realization of parallel programs.

Keywords : multicore, parallelism, Qt, Deriche.