République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de La Recherche Scientifique

Université Mouloud MAMMERI de Tizi-Ouzou
Faculté du Génie électrique et Informatique
Département Informatique

Mémoire

De fin d'études

En vue de l'obtention du diplôme de Master en informatique

Option « Système Informatique »

Thème

Traitement parallèle sous Paratask de java Application : Algorithme de filtrage de R.Deriche

Proposé et dirigé par :

M^r^ L.DJEMA

Réalisé par :

M^elle^ MEZIANE Samira
M^elle^ OUACHEHI Kahina

20011/2012

Remerciements

C'est avec humilité et gratitude que nous reconnaissons ce que nous devons :

Nous commençons par remercier et rendre grâce à Dieu le tout puissant de nous avoir donné le courage et la volonté de mener à bon terme ce travail.

Ainsi nous tenons à remercier notre promoteur Mre : DJEMA qui a su nous guider et nous orienter.

Nous tenons aussi à remercier tous les enseignants du département informatique

Nous tenons à remercier également nos enseignants de différentes phases d'enseignement.

Nous tenons à remercier également notre collègue Khaled BELKACEM pour son aide.

Nous tenons à remercier tous ceux qui nous ont aidés de prés ou de loin à la réalisation de ce travail, il n'y a pas de place pour les citer que dans le cœur.

A tous, nous exprimons notre vive gratitude.

Dédicaces

Dieu merci, Dieu merci, Dieu merci… !

Je dédié ce modeste travail :

Tout d'abord à celui qui n'a jamais quitter mes pensés, mon père, qu'il repose en paix ;

A mon adorable et bien aimée mère, que dieu nous la garde ;

A mes trois frères Jugurtha,Gaya et Massi ;

A mes très chères et bien aimées soeurs Dyhia et Lynda ;

A celui qui a toujours était à mes cotés et que j'aime de tout cœur Samir ;

A mes tantes, oncles et grands parents ;

A tous mes amis et collègues Anis, Takfa, Redha, Saber et Slimane ;

A mon adorable amie Lyly ;

Sans oublier, mon binôme et amie Samira, ainsi que toute sa famille

Kahina

Dédicaces

Dieu merci, Dieu merci, Dieu merci… !

Je dédié ce modeste travail :

A ma chère maman et mon adorable père ; que dieu nous les gardent ;

A mes adorables frères : Elghani et Ghilas

A mes très chères sœurs : Dalouche et Fifi

A ma très chère Doho ;

A mon adorable Lyly qu'elle reste toujours comme elle est ;

A toute ma famille : cousins ; cousines ; tantes et oncles ;

A mes amis : Anis, Takfa, Slimane, Redha et Saber ;

A tout mes amis et camarades ;

A mon binômes Kahina et toute sa famille ;

A tous ceux que je connaisse de prés ou de loin ;

Samira

Sommaire

Chapitre 3 : les outils de parallélisme

Chapitre 4 : Réalisation

Liste des figures

La première solution qui nous vient à l'esprit dès qu'un besoin de puissance s'impose est le calcul parallèle. En effet, le parallélisme est devenu une nécessité pour résoudre principalement des problèmes scientifiques tels que la synthèse des images, la simulation de vol, et l'animation vidéo; le plus rapidement possible. Le principe de base de ce dernier est d'utiliser plusieurs ressources à titre d'exemple les processeurs ; qui fonctionnent concurremment pour accroître la puissance de calcul pour la résolution d'un même algorithme pour un problème donné. Son objectif est non seulement de résoudre les problèmes le plus rapidement, mais aussi de pouvoir résoudre des problèmes de plus grande taille.

Ce mémoire est fondé sur quatre chapitres, le premier est une généralité sur le parallélisme ; le second explique comment concevoir un code parallèle à partir d'un autre séquentiel ; le troisième nous donne une étude des outils mis en œuvre dans le but de paralléliser un code ; et enfin le dernier qui est consacré pour mieux expliquer l'outil (la bibliothèque) parallèle de java « ParaTask » utilisée durant le développement de notre application.

L'objectif de notre projet est de mettre en œuvre le parallélisme en pratique, est ce en créant une application desktop, qui permet de détecter les contours d'une image médicale.

I- Introduction :

La fin des années 80 a vu se développer une nouvelle classe d'ordinateurs comportant plusieurs processeurs fortement couplés, qui travaillent en parallèle et coopèrent à la solution d'un même problème. La raison qui a conduit à ce genre de machines est avant tout le besoin croissant de rapidité de calcul dans les applications scientifiques modernes.

Autrement dit, on utilise plusieurs ressources de calcul pour exécuter un programme, Ce programme s'exécute dans de multiples CPUs, il est subdivisé en un ensemble de parties discrètes qui s'exécutent d'une façon concurrente, chaque partie est encore subdivisée en un ensemble d'instructions et ces instructions s'exécutent simultanément dans des CPUs différentes.

Pour une exécution plus rapide, il est naturel d'utiliser davantage de processeurs. Les performances d'un processeur séquentiel ne peuvent d'ailleurs pas augmenter à l'infini pour des raisons physiques. De plus, le prix d'un processeur séquentiel accroît en fonction de sa puissance. En première approximation, il est donc plus avantageux du point de vue coût, de relier plusieurs processeurs lents que d'en acheter un très rapide.

Pour faire face à ces problèmes, les ingénieurs ont pensé aux systèmes parallèles qui sont devenus une norme, qu'il s'agisse des systèmes de calcul ultra-haut performances, des ordinateurs personnels ou des serveurs web.

Dans ce qui suit nous allons introduire la notion des architectures parallèles, celle du parallélisme et la programmation parallèle.

II- Concepts et terminologies sur les architectures parallèles:

II-1- L'architecture Von Neumann:

L'architecture, dite architecture de Von Neumann, est un modèle pour un ordinateur qui utilise une structure de stockage unique pour conserver à la fois les instructions et les données requises ou générées par le calcul.

Le modèle de Von Neumann dit modèle séquentiel, donne les quatre composants essentiels qui constituent un micro-processeur. Il décrit également les interactions entre ces différents composants. Les éléments de base sont la mémoire et le processeur, lui-même constitué d'une unité de contrôle et d'une unité de traitement. L'unité de contrôle lit dans la mémoire les instructions à exécuter, puis envoie des ordres à l'unité de traitement. Celle-ci effectue alors les opérations demandées sur le flux de données en provenance de la mémoire. Le flux traité est ensuite réécrit en mémoire. [Web 00]

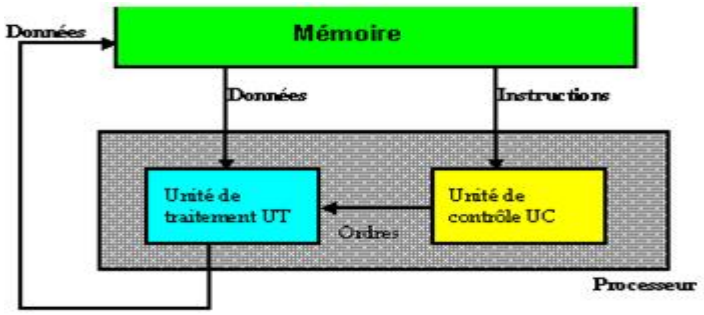


Figure1.1 :l'architecture Von Neumann.

II-2- Classification des ordinateurs parallèles : [web 00]

En 1969, M.J. Flynn proposa un système de classification des différents modèles de fonctionnement envisageables pour un calculateur. Cette classification est toujours utilisée de nos jours. Cette dernière est fondée sur deux critères suivant:

- la machine a-t-elle un ou plusieurs flux de données? (Single Data stream ou Multiple Data stream).
- la machine a-t-elle un ou plusieurs flux d'instructions? (Single Instruction stream ou Multiple Instruction Stream).

La quasi-totalité des ordinateurs disponibles sont conçus sur la base d'une de ces architectures :

	Flot d'instructions	
Flot de données	unique	multiple
unique	SISD	MISD
multiple	SIMD	MIMD

Figure 1.2 : Classification des modèles de parallélisme, proposée par M.J Flynn.

a- Single Instruction Single Data (SISD) :

Le modèle purement séquentiel est noté SISD (Single Instruction Single Data). Hérité du modèle d'architecture Von Neumann, Cette catégorie correspond aux machines, pour lesquelles chaque opération s'effectue sur une donnée à la fois.

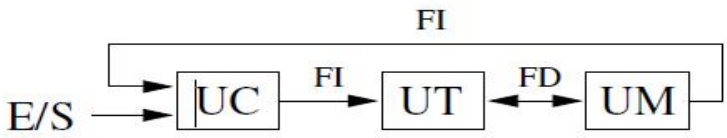


Figure1.3 : Architecture SISD.

b- Multiple Instruction, Single Data (MISD):

L'architecture MISD, conserve la même structure que celle de Von Neumann. La différence est que les unités de traitement et de contrôle sont découpées en étages, chargés chacun d'une partie des opérations à effectuer. Le facteur d'accélération est donc égal au nombre d'étages des unités

Elle correspond aux systèmes capables d'exécuter plusieurs instructions sur la même donnée durant le même cycle d'horloge.

Cette architecture a juste un intérêt théorique puisque aucun système pratique n'a été construit à l'aide de cette organisation.

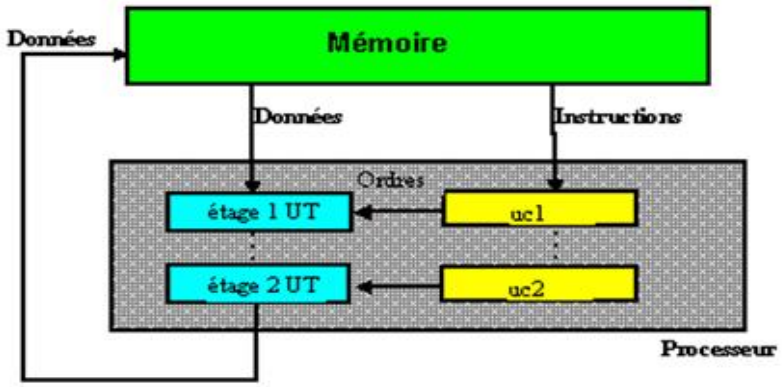


Figure1.4: Multiple Instruction, Single Data.

c- Single Instruction, Multiple Data(SIMD):

Dans le modèle SIMD (Single Instruction Multiple Data), seules les unités de traitement sont dupliquées, elles sont commandées par une unité de contrôle unique ce qui entraîne que chaque processeur de cette architecture exécute la même instruction à chaque cycle d'horloge, mais les données traitées sont différentes.

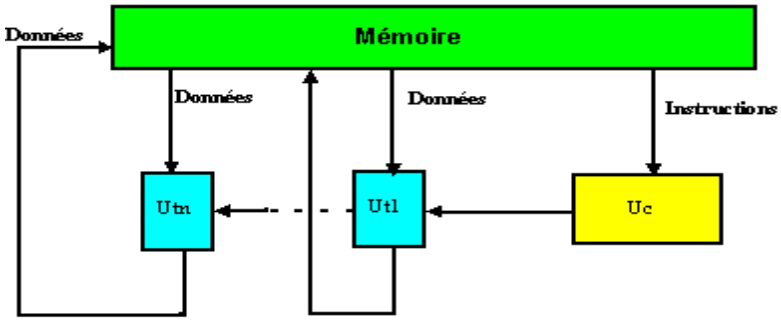


Figure1.5 : Single Instruction, Multiple Data.

d- Multiple Instruction, Multiple Data(MIMD) :

C'est dans cette catégorie que l'on trouve le plus de systèmes parallèles. A chaque cycle d'horloge, chaque processeur exécute une instruction différente sur une donnée différente.

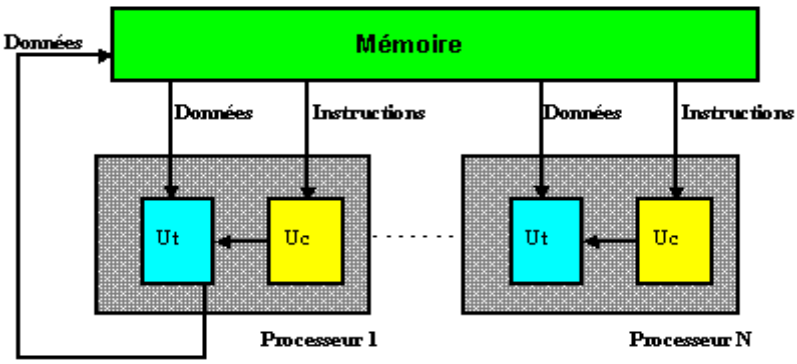


Figure1.6 : Multiple Instruction, Multiple Data.

Le schéma ci-dessus représente l'architecture d'un système MIMD à mémoire commune, sur ce principe les différents processeurs communiquent entre eux grâce à cette mémoire (opération de Lecture/Ecriture aux adresses mémoire).

II-3- Les architectures mémoire d'un ordinateur parallèle :

II-3-1-La mémoire partagée :

1-Définition :

Dans ce cas, plusieurs processeurs ont accès à la même mémoire physique. Ils peuvent opérer avec elle de manière indépendante et les changements faits par l'un des processeurs sont immédiatement visible par les autres. [CP 2004]

Les machines à mémoire partagée permettent de réaliser le parallélisme de données et de contrôle. Le programmeur n'a pas besoin de spécifier la distribution des données sur chaque processeur. Il définit seulement la partie du programme qui doit être parallélisée (directives) et doit gérer les synchronisations. [Web 01]

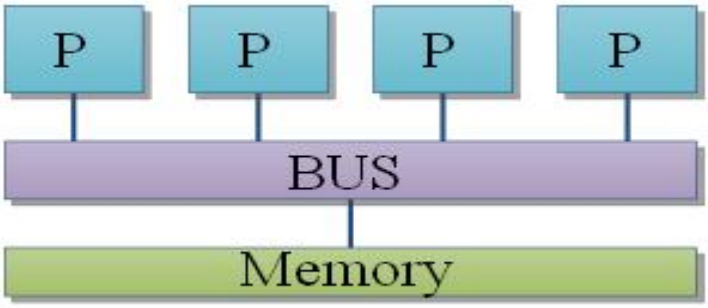


Figure 1.7 : La mémoire partagée.

2- Caractéristiques :

- Plusieurs processeurs avec des horloges indépendantes.
- Une seule mémoire commune à tous les processeurs.
- Programmable par le standard portable OpenMP[1].
- Plusieurs processeurs peuvent travailler indépendamment mais partager la même mémoire.

Les machines à mémoire partagées peuvent être regroupées en deux classes par rapport au temps d'accès: UMA et NUMA.

3- Classes des machines à mémoire partagée: [BL] [CP 2004]

• Uniform Memory Access (UMA):

C'est une architecture à mémoire partagée utilisée dans les ordinateurs parallèles. Tous les processeurs dans le modèle UMA partagent la mémoire physique, où l'accès à cette dernière est le même pour chaque processeur, on parle alors d'accès équitable.

Il y a aussi, dans ce type d'architecture mémoire, ce que l'on appelle un contrôleur de cache (CC-UMA) qui s'assure que la mémoire cache de chaque processeur est cohérente avec les

[1] Paradigme de parallélisation pour l'architecture à mémoire partagée basé sur des directives à insérer dans le code source (C, C++, Fortran).

données présentes en mémoire centrale. Ce type de mémoire est présent dans la plupart des systèmes SMP[2] (Symetric MultiProcessor).

L'architecture UMA est convenable pour les applications partagées par plusieurs utilisateurs, elle peut être utilisée pour accélérer l'exécution des grands programmes dans des applications à temps critique.

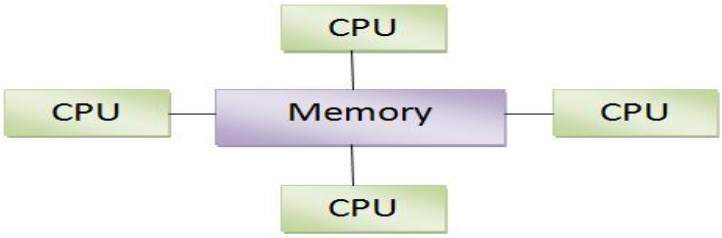


Figure 1.8: Uniform Memory Access (UMA).

- **Non -Uniform Memory Access (NUMA):**

Les mémoires à accès non uniformes (NUMA : Non Uniform Memory Access) où chaque processeur peut accéder à la mémoire indépendamment des autres. Ils peuvent donc être de types et de vitesses différentes. Comme pour l'architecture UMA, on peut y rajouter un système de cohérence de cache, mais l'implémentation de ce dernier est compliquée par ces accès non uniformes.

Ce type de mémoire présente l'avantage de permettre un partage immédiat des données, facilitant la programmation. Mais cette solution coûte chère, ce qui limite le nombre de processeurs que l'on peut ajouter sur une même mémoire. De plus, les mécanismes de cohérence de cache sont coûteux en performance et plus on ajoute de processeurs, plus ce type de mécanisme devient indispensable. Si on ajoute le fait que le débit de la mémoire est limité, on voit bien que l'accroissement des performances ne suit pas linéairement le nombre des processeurs. [CP 2004]

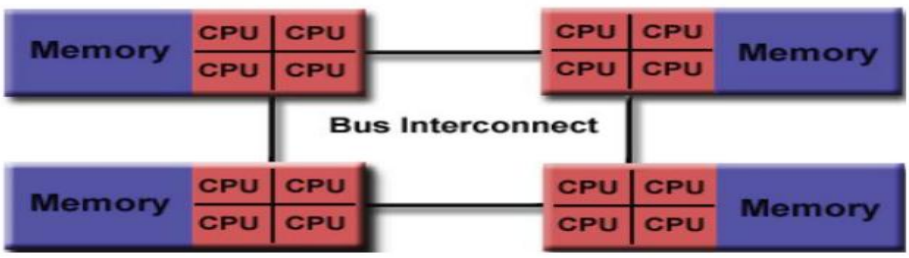


Figure 1.9: Non-Uniform Memory Access (NUMA).

[2] Ce principe est apparu au milieu des années 1970, système multiprocesseur distribuant symétriquement les tâches entre différents processeurs partageant une mémoire commune, en s'assurant qu'ils ne vont pas se mettre à écrire tous à la même adresse en même temps.

4- Avantages et inconvénients de l'architecture : [BL]

v Avantage :

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

v Inconvénients :

- La portabilité entre la mémoire et les CPUs : l'utilisation de plusieurs CPUs peut augmenter le trafic dans le chemin partagée mémoire -CPU, et peut augmenter aussi le trafic associé pour la gestion de la mémoire cache pour les systèmes Cache Curency.

- La responsabilité du programmeur est la synchronisation qui assure un accès correct à la mémoire global.
- La conception et la production des machines à mémoire partagée sont difficiles et couteuses.

II-3-2- La mémoire distribuée:

1- Définition :

Dans ce cas, chaque processeur possède sa propre mémoire. La modification par l'un des processeurs dans sa propre mémoire n'a pas d'influence directe sur celle des autres processeurs. Cela suppose donc de mettre en place une communication explicite entre les processeurs (souvent par l'intermédiaire d'un réseau)**[CP 2004]** .Ce réseau peut être soit interne à la machine et géré au niveau matériel (NUMA) soit externe à un nœud de calcul (Ferme de calcul[3]) et partagé au niveau logiciel (MPI[4]). **[Web 01]**

Les communications entre processeurs sont réalisées par l'appel des fonctions de bibliothèque standard : PVM[5] (Parallel Virtual Machine) ou MPI (Message Passing Interface).

Il est préférable de développer avec MPI qui est plus stable et qui est portable sur les machines parallèles actuelles. **[Marchand MPI]**

[3] Dites aussi grappes de serveurs sont un procédé peu coûteux, résidant dans la mise en place de plusieurs ordinateurs en réseau qui vont apparaître comme un seul ordinateur ayant plus de capacités (plus puissant, etc.) , très utilisé pour les calculs parallèles.
[4] conçue en 1993-94, est une norme définissant une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages.
[5] Bibliothèque de communication (langages C et Fortran) pour machines parallèles et réseau d'ordinateurs

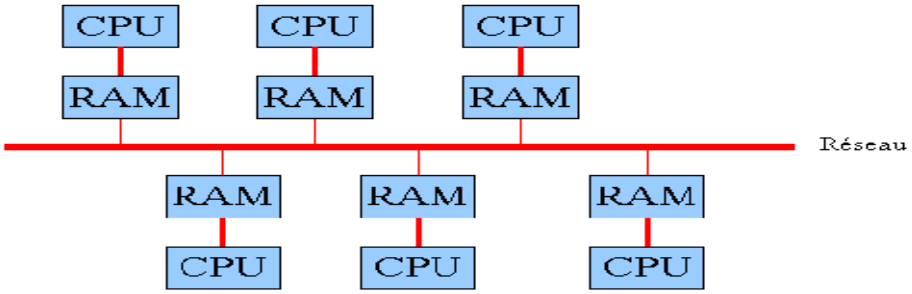


Figure 1.10 : La mémoire Distribuée.

2- Avantages et inconvénients de l'architecture: [BL]

v Avantages

- L'augmentation du nombre de processeur augmente la mémoire.
- Chaque processeur peut accéder directement à ces mémoires.

v Inconvénients :

- Le programmeur est responsable des détails de transmission de données entre les processeurs.
- Dans cette organisation mémoire il est difficile d'adresser les structures de données existantes dans la mémoire global.

II-3-3- La mémoire Hybride (Distribuée-partagée):

Ce dernier type de mémoire est un mélange des deux premiers. Dans cette architecture, il y a plusieurs groupes de processeurs partageant de la mémoire qui communiquent grâce à un réseau. Cela permet, dans une certaine mesure, de tirer les avantages des deux architectures précédentes et d'en réduire les inconvénients. **[CP 2004]**

Le composant mémoire partagé est une machine SMP cache cohérent, les processeurs dans SMP peuvent avoir une mémoire machine globale.

Le composant mémoire distribué est un réseau de SMPs, les SMPs savent ce qui ce passe dans leur propre mémoire seulement et pas dans la mémoire des autres SMPs, donc un réseau de communication est obligatoire pour envoyer des données d'un SMP à un autre. Ce type d'architecture augmente la vitesse d'exécution. **[BL]**

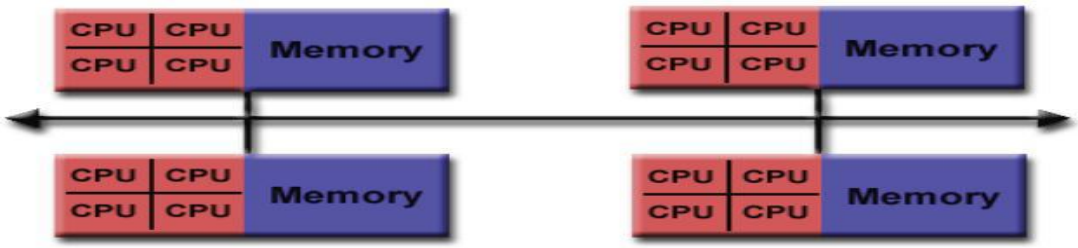


Figure 1. 11: La mémoire Hybride (Distribuée-partagée).

III- Le parallélisme :

III-1- Les classes du parallélisme : [Belkacem 2011]

Le parallélisme a pour but d'augmenter la vitesse d'applications en multipliant les ressources de calcul d'un système. Pour en bénéficier, l'application doit être pensée en termes de tâches indépendantes qui pourront être exécutées de manière concurrente par les différentes unités de calcul. Deux classes de parallélismes sont définies :

Ø Le parallélisme asynchrone :

Dans le parallélisme asynchrone, le problème est divisé en plusieurs sous problèmes pour les distribués aux différents processeurs. Un programme parallèle asynchrone contient plusieurs threads de contrôle. Si les sous problèmes ne sont pas complètement indépendants, ils doivent échanger des informations et doivent être synchronisés manuellement.

Les processus exécutent plusieurs sous problèmes, chaque sous problèmes est divisé en plusieurs instructions comme les expressions arithmétique et ceci rend le cout de la synchronisation plus élevé.

Ø Le parallélisme synchrone :

Pour le parallélisme synchrone, les processeurs sont indépendants, les processus utilisés pour effectuer une tâche sont contrôlés par un processeur central, le programme parallèle synchrone a un seul thread de contrôle, et ceci simplifie l'intégration d'autres processeurs. De cette façon le parallélisme synchrone peut intégrer plus de processeurs que le parallélisme asynchrone.

La synchronisation entre les différents processeurs se fait implicitement à chaque étape sans l'intervention du programmeur.

III-2- Modèles de parallélisme : [Vollaire 1997]

Il existe deux notions de parallélisme: **homogène** et **hétérogène**. Il faut noter qu'en calcul scientifique, la plupart des applications utilisent le parallélisme homogène, tandis que la programmation parallèle hétérogène est surtout présente au niveau du système

d'exploitation (UNIX), dans les applications à temps réel et dans les protocoles de communications .

Ø Parallélisme hétérogène :

Le parallélisme réside dans l'exécution en parallèle de tâches (instructions différentes). Ce type de programmation consiste à séparer les actions à réaliser, donc les fonctions du programme. Cette programmation est appelée programmation concurrente[6] par opposition à la programmation parallèle. Elle est présente dans les algorithmes de type maître/esclave, producteur/consommateur. Les outils de base d'une telle programmation sont le thread d'exécution, le verrou, le sémaphore ou encore la barrière. Du fait de la nature asynchrone de cette programmation, leur utilisation peut s'avérer très complexe.

Ø Parallélisme homogène :

Dans ce type de programmation, une même tache est appliquée à des données différentes (Single Program, Multi Data). Le parallélisme s'applique donc sur les données et cette programmation est dite parallèle.

III-3- Les techniques d'utilisation : [Vollaire1997] [Belkacem 2011]

Il existe trois façons d'exploiter le parallélisme d'une application donnée: parallélisme de contrôle, de flux et sur les données.

Ø Parallélisme de contrôle :

Ce parallélisme met en œuvre l'exécution simultanée de plusieurs tâches plus ou moins indépendantes. Généralement, chaque processus est affecté à un processeur de la machine pour toute la durée de son exécution. Une application réelle, à cause de certaines dépendances, nécessite des synchronisations entre les tâches et parfois des mises à jour de données communes.

Ø Parallélisme de flux :

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

[6] C'est un style de programmation tenant compte, dans un programme, de l'existence de plusieurs piles sémantiques. Ces piles peuvent être appelées threads, processus ou tâches. Elles sont matérialisées en machine par une pile d'exécution et un ensemble de données privées. Les threads disposent d'une zone de mémoire partagée alors que les processus sont strictement isolés.

programmation parallèle est égale au nombre d'étages du pipe-line. Il y' a dépendance entre les instructions lorsqu'elles référent au même mot mémoire.

Ø Parallélisme de données :

Ce parallélisme est basé sur l'application d'une même suite d'opérations sur des données différentes. Ce type de programmation est surtout rencontré dans des applications utilisant du calcul matriciel. La même action est donc répétée sur différents éléments de la structure traitée. Ce type de programme peut se dérouler de manière synchrone, sur des architectures SIMD par exemple, ou asynchrone, sur des architectures MIMD nécessitant une duplication du code sur chaque processeur et éventuellement une lecture des données par processeur.

III-4- Les dépendances de données : [Vollaire1997]

Ce problème est rencontré sur les calculateurs parallèles à mémoire partagée où le parallélisme est réalisé par distribution des itérations des boucles sur les différents processeurs. Il existe une dépendance de données si une boucle modifie une case mémoire à une itération et que, lors d'une autre itération, cette case mémoire est lue ou écrite. Par contre il n'y a pas dépendance si une seule itération utilise une variable ou encore si plusieurs itérations lisent une variable sans la modifier. Les dépendances sont la cause majeure de la déficience des compilateurs à paralléliser certaines boucles. Il est donc important pour le programmeur de savoir les identifier quand il se retrouve confronté à une boucle non optimisée par le compilateur. Les dépendances plus significatives sont :

Ø Dépendances de flux :

- **Dépendance avant:**

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

Dans cet exemple, a(i) est utilisé dans une itération et modifié dans la suivante. Il est toutefois possible de paralléliser cette boucle soit en synchronisant la boucle par affectation de la bonne valeur avant sa modification, soit en insérant un tampon et en effectuant deux boucles. La première remplit le tampon avec 'a' et la deuxième met à jour 'a' avec le tampon.

- **Dépendance arrière:**

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

a(i) est défini lors d'une itération et est modifié à la suivante. Ce type de dépendance est difficile à paralléliser et est très répandue. La seule manière de paralléliser cette boucle consiste à synchroniser les itérations de façon à garantir l'ordre d'exécution des écritures / lectures.

- **Réduction:**

```
For(i=0,i<n,i++)
  {x=x+a(i)}
```

x dépend de sa valeur à une autre itération, cette opération sur x est appelée une réduction. Cette boucle peut être parallélisée en ajoutant un tableau tampon de longueur égale au nombre de processeurs. Chaque processeur effectue une addition partielle de a sur le terme du tableau tampon correspondant à son numéro. Une autre boucle permet de réaliser la sommation totale à partir de tous les termes du tableau tampon.

Ø Dépendance de contrôle:

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

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

III-5- Supports de parallélisme : [Deghilage,1992]

Un programme parallèle est un ensemble de processus séquentiels s'exécutant d'une façon simultanée. Ces processus coopèrent ou entrent en compétition lors de l'exécution, en partageant des variables ou en échangeant des informations. Ces variables partagées doivent être accessibles par un seul processeur à la fois, pour maintenir la consistance des données, l'accès à ces variables doit être contrôlé. Il existe deux types de contrôle : la synchronisation entre les processeurs et la cohérence des données dans les différents éléments de la hiérarchie mémoire.

III-5-1- Les mécanismes de synchronisation :

Un système multiprocesseur doit posséder des mécanismes de synchronisation à trois niveaux différents : le matériel (mémoire et processeurs), les systèmes d'exploitation et le programme utilisateur. Chacun de ces niveaux dispose de plusieurs moyens de synchronisation dont le choix dépend de l'application.

La mise en œuvre d'une machine utilisant plusieurs processeurs pose de nombreux problèmes à un système d'exploitation. Chaque processeur est relié à la mémoire qui doit être cohérente pour tous les processeurs. La synchronisation des différents constituants matériels d'une machine multiprocesseur peut se faire par matériel ou par logiciel.

- **Synchronisation avec le logiciel :**

La plupart des solutions logicielles sont conçus pour un nombre donné de processus et sont donc inappropriées pour un multiprocesseur à usage générale et l'efficacité des mécanismes de synchronisation affecte la qualité de l'environnement de programmation parallèle.

- **Synchronisation avec le matériel :**

L'implémentation des synchronisations avec le matériel augmente le cout global du système. Il est donc important de trouver le moyen matériel le plus économique pour réaliser les synchronisations.

L'élément fondamental consiste à pouvoir lire et modifier une variable d'une façon atomique, c'est-à dire en un cycle avec la garantie qu'aucune interruption ou exception de priorité n'intervient pendant cette opération.

Il existe trois types d'implémentations :

- Basée sur les sémaphores.
- Basée sur les moniteurs
- Basée sur les passages de messages.

III-5-2- La cohérence des caches :

L'approche la plus classique pour implémenter une mémoire globale physiquement partagées, consiste à relier les processeurs et les mémoires par un bus commun. Cette organisation risque de limiter le nombre de processeurs qui partagent le bus.

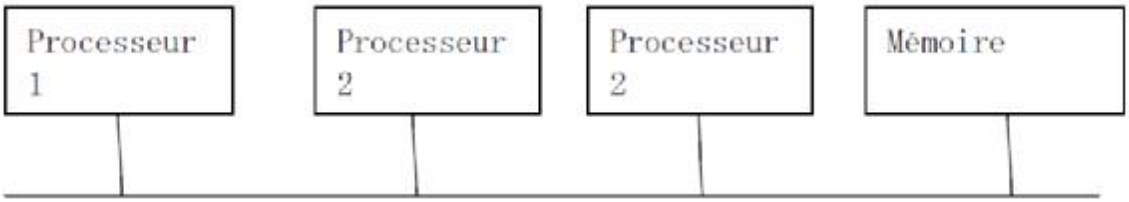


Figure 1.12 : Partage de la mémoire par plusieurs processeurs sans la cohérence des caches.

Le premier moyen utilisé pour réduire le trafic sur le bus consiste à joindre un cache à chaque processeur.

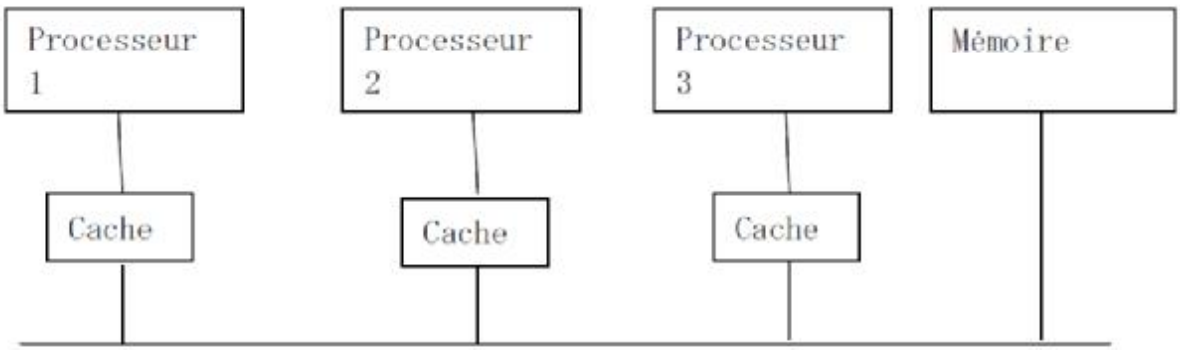


Figure 1.13 : Partage de la mémoire par plusieurs processeurs avec la cohérence des caches.

Admettons que les processeurs 1 et 2 lisent un mot dans la mémoire. Une copie sera alors dans chaque cache 1 et 2. Si le processeur 1 modifie la valeur de ce mot, modifié dans le cache 1, le processeur 2 doit être averti de ce changement pour utiliser la nouvelle valeur du mot, conformément à la définition de mémoire partagée. Ceci pose le problème de la cohérence des caches, qui est incontournable si l'on veut pouvoir mettre plus de 2 ou trois processeurs sur le même bus.

III-6- Quelques termes liés à la programmation parallèle :

- **Tâche :** C'est une partie d'une application exécutée de manière indépendante par une unité de calcul. Elle a son propre espace de données et a accès à l'ensemble des mémoires du système.

- **Tâche parallèle :** C'est une tâche qui peut être exécutée par plusieurs processeurs.

- **Exécution en série :** C'est l'exécution instruction par instruction (séquentielle) d'un programme.

- **Exécution parallèle :** C'est l'exécution de plusieurs tâches à la fois, chaque tâche exécute la même instruction ou une instruction différente à la fois.

- **SMP (Symmetric Multi-Processor) :** Système multiprocesseur distribuant symétriquement les tâches entre les différents processeurs, partageant une mémoire commune, en s'assurant qu'ils ne vont pas se mettre à écrire tous sur la même adresse en même temps.

- **Compare-and-swap :** En français Diviser pour régner, une technique algorithmique consistant à diviser un problème de grande taille en plusieurs sous-problèmes analogues. L'étape de subdivision est appliquée récursivement.

- **Thread worker:** Un thread worker est un thread utilisé pour gérer des requêtes qui se présentent comme une file de travail (tâches) associée à ce thread. Opère dans une boucle qui récupère les requêtes de la file de travail (aussi attendre une requête si nécessaire), traite la requête appropriée et retourne à la liste de travail pour une requête suivante.

IV- Conclusion :

Ce chapitre est divisé en deux parties ; la première consiste à introduire et à définir certaines notions de base relatives aux architectures parallèles, et la seconde à pour but de présenter la notion de parallélisme, ces différents types ainsi que les mécanismes de synchronisation dans les systèmes parallèle, et pour finir quelques notions de base liées à la programmation parallèle.

Le parallélisme pose de nouveaux problèmes au niveau de l'algorithmique à utiliser pour répartir intelligemment le travail sur plusieurs unités de traitement. Finalement des problèmes se posent au niveau des langages de programmation à adopter et des opérations de gestion, de placement et d'ordonnancement des différentes tâches qui doivent être exécutées simultanément, et c'est ce qu'on verra dans le chapitre qui suit.

I-Introduction :

De nos jours, les avancées technologiques orientent les concepteurs de systèmes vers une augmentation du nombre de ressources pour effectuer un traitement, plutôt qu'augmenter la puissance d'une ressource pour exécuter ce même traitement.

La construction d'un algorithme parallèle pour un problème donné, est beaucoup plus complexe que la construction d'un algorithme en séquentiel du même problème, car elle demande la prise en compte de plusieurs facteurs qui sont par exemple : la partie du programme qui peut être traitée en parallèle, la manière de distribuer les données, les dépendances des données, la répartition de charges entre les processeurs, et les synchronisations entre ces derniers. Il y a essentiellement deux méthodes pour construire un algorithme parallèle, l'une consiste à détecter et à exploiter le parallélisme inhérent dans un algorithme séquentiel déjà existant, l'autre à inventer un nouvel algorithme dédié au problème donné. Dans ce chapitre, nous allons présenter la notion compilateurs parallèles, puis nous passerons au processus de conception de programmes parallèles, et pour terminer un ou deux exemples d'application de ce processus.

II- Les compilateurs parallèles :

Les compilateurs parallèles travaillent avec deux façons différentes :

II-1- Le parallélisme automatique : [Web 01]

Le parallélisme automatique est une étape de la compilation d'un programme, qui consiste à transformer un code source écrit pour une machine séquentiel en un exécutable parallélisé pour ordinateur à système SMP.

Son objectif est, de simplifier et de réduire la durée du développement des programmes parallèles, qui sont plus compliqués à écrire que les programmes séquentiels mais permettant des gains de vitesse sur les machines parallèles.

II-2- Dirigé par le programmeur : [Web 02]

En utilisant des directives de compilation, le programmeur définit comment le compilateur doit paralléliser le code, peut aussi utiliser un niveau de parallélisme automatique.

Si des contraintes de budget ou de temps se présentent, le parallélisme automatique est la bonne solution, mais des inconvénients peuvent se présenter tels que :

- De faux résultats peuvent se produire.
- Dégradation de performance.
- Moins flexible que le parallélisme manuel.
- Généralement limité pour paralléliser les codes qui contiennent des boucles.

- Beaucoup de code reste non parallèle si l'analyse trouve le code complexe.

III- Concevoir une application parallèle : [web 04]

Dans cette partie, nous allons présenter une démarche à suivre afin de concevoir un programme parallèle à partir d'un programme séquentiel déjà existant.

III-1- Comprendre le problème ou le programme:

Afin de tirer les meilleures performances d'une application parallèle, il faut commencer par bien comprendre le problème.

En effet, il faut tout d'abord déterminer si une exécution parallèle de l'application peut apporter un gain de performance. Pour cela, il faut qu'il y ait une certaine indépendance dans le traitement des données. Soit le traitement sur une partie des données est indépendant du traitement des autres, soit il existe des tâches qui peuvent être traitées indépendamment des autres. Par exemple, le calcul de la suite Fibonacci (F (k+2) = F (k+1) + F (k), chaque élément est égal à la somme des deux précédents) ne peut pas être distribué à cause de la trop grande dépendance des calculs. Il faut attendre le calcul des deux termes précédent pour pouvoir calculer le terme courant.

Il faut aussi trouver les parties les plus gourmandes en termes de temps de calcul de l'application. En effet, c'est sur ces parties que le travail de mise en parallèle sera le plus efficace en terme de performance.

III-2- Le partitionnement ou décomposition :

Le partitionnement est la première étape d'écriture d'un programme parallèle. Il consiste à découper le problème, soit en terme de données indépendantes, soit en terme de fonctionnalités. Ces deux approches de décomposition sont respectivement appelées : la **décomposition du domaine** et la **décomposition fonctionnelle**.

C'est de ce partitionnement que va dépendre le gain apporté par la mise en parallèle du calcul. Chaque partie du calcul considérée comme indépendante pourra alors être exécutée par un processeur différent.

Ø La décomposition du domaine : [web 02]

Dans l'approche de décomposition de domaine du partitionnement des problèmes, nous cherchons d'abord à décomposer les données associées à un problème. Si possible, nous divisons ces données en de plus petits morceaux de tailles à peu près égales. Ensuite, nous partitionnons les calculs devant être réalisés, généralement en associant chaque opération aux données qu'elle utilise. Ce partitionnement réalise un certain nombre de tâches, chacune comprenant des données et un ensemble d'opérations sur ces données.

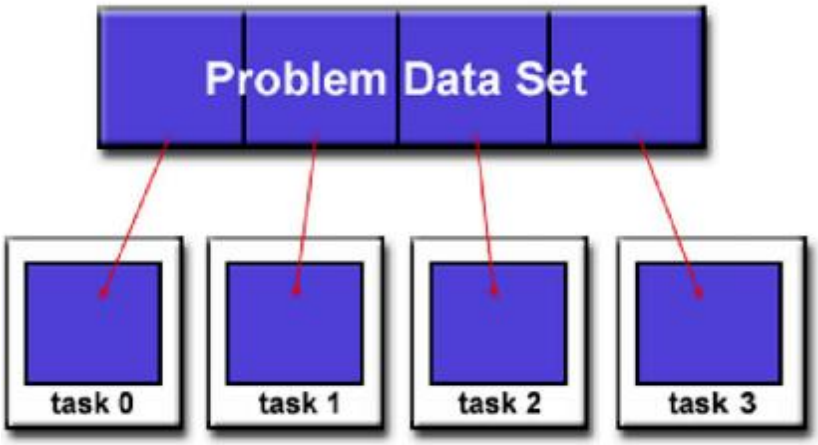


Figure 2.1 : Décomposition du domaine.

On peut décomposer les données par bloc, cyclique ou en combinant les deux comme le montre la figure suivante :

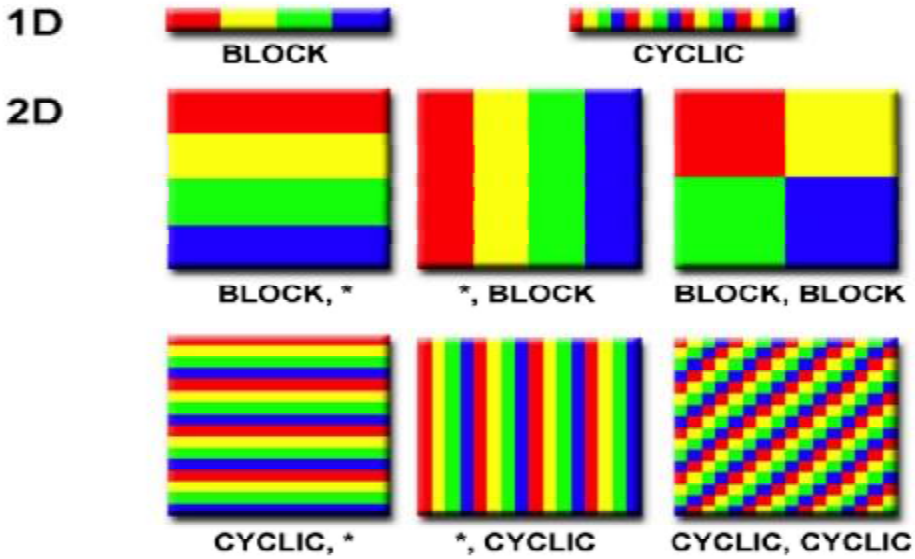


Figure 2.2 : Décomposition par bloc et décomposition cyclique des données

Ø La décomposition fonctionnelle : [Web 03]

La décomposition fonctionnelle repose sur la reformulation des calculs, pour avoir des résultats décomposables et donc parallélisables. Dans cette approche, on se base sur le traitement qui va s'exécuter plutôt que les données manipulées par ce traitement.

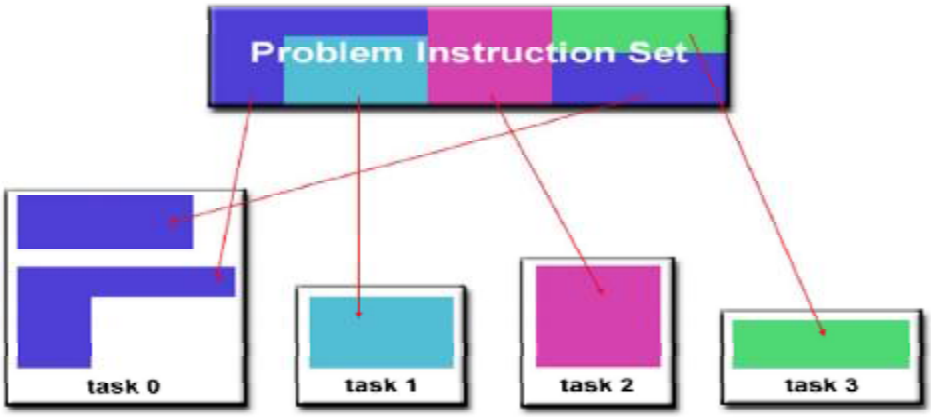


Figure 2.3 : Décomposition fonctionnelle.

III-3- La communication :

La communication entre les processeurs dépend de la tâche à effectuer. En effet, lorsque le traitement d'une donnée est complètement indépendant des autres, il n'est pas nécessaire de mettre en place un système de communication. Malheureusement la grande majorité des calculs, pour être bien mené, ont besoin d'une partie des résultats des autres processeurs voisins. Il est donc nécessaire de diffuser les résultats vers ceux qui en ont besoin.

Une communication est considérée comme un canal qui relie les tâches communicantes, dans lequel ces tâches peuvent envoyer et recevoir des messages. Au début on définit une structure canal qui relie les tâches, les tâches qui consomment les données (consommateurs) avec les tâches qui fournissent les données (producteurs). Ensuite, on spécifie le message à envoyer sur le canal.

Ø Les différents types de communication :

- **La communication synchrone :**

Implique seulement les tâches qui exécutent l'opération de communication. Quand une tâche exécute une opération de communication elle coordonne avec d'autres tâches qui participent dans la communication. Par exemple une tâche qui exécute une opération d'envoie des données doit d'abord avoir l'autorisation de la tâche qui va recevoir ces données. Les communications synchrones sont plus lentes et sont généralement bloquantes, c'est à dire que les deux tâches engagées dans la communication doivent attendre la fin de la communication pour continuer leur exécution.

- **La communication asynchrone :**

Dans une communication asynchrone, la tâche qui fournit la donnée (producteur) n'est pas capable de savoir quand les autres tâches (consommateurs) ont besoin de la donnée, donc les

consommateurs doivent récupérer d'une façon explicite les données des tâches productrices. La communication asynchrone est utilisée quand un traitement est structuré en un ensemble de tâches qui doivent périodiquement accéder aux données partagées pour la lecture et l'écriture. Les communications asynchrones sont la plupart du temps non bloquantes. Quand une tâche veut envoyer un message à une autre, elle envoie le message et peut immédiatement reprendre son exécution sans se soucier, quand l'autre tâche recevra le message. C'est là le principal avantage des communications asynchrones.

III-4- La synchronisation :

Il existe différents types de synchronisations :

- Les barrières impliquent l'ensemble des tâches de l'application. Pour passer une barrière, il faut que toutes les tâches de l'application aient atteint cette barrière.
- Les verrous et sémaphores impliquent un nombre quelconque de tâches. Ils sont utilisés principalement pour protéger une donnée ou une section critique du code. Lorsqu'une tâche obtient le verrou, le système doit garantir qu'elle est la seule à l'avoir et qu'elle peut, en toute sécurité, accéder à la donnée ou rentrer dans la section critique. Les autres tâches cherchant à obtenir ce verrou pendant ce temps seront bloquées en attente du verrou.
- Les communications synchrones peuvent aussi jouer le rôle d'élément de synchronisation.

III-5- La dépendance des données :

Il y a dépendance entre deux parties d'une application lorsque l'exécution de l'une affecte le résultat de l'autre. Une dépendance de donnée entraîne une utilisation de la valeur d'une même variable par des tâches différentes. Ces dépendances sont très importantes en ce qui concerne les applications parallèles puisque c'est l'un des principaux freins au développement d'applications parallèles.

Elles peuvent être gérées soient par la communication des données entre les processeurs lors de points de synchronisation (dans le cas de mémoires distribuées) soit par la synchronisation des lectures / écritures (dans le cas de mémoires partagées).

III-6- La répartition des charges : **[web 05]**

La répartition de charge parfois appelée partage de charge (*Load Sharing*) consiste à utiliser au maximum les ressources disponibles d'une machine. Le cas idéal est d'occuper en permanence ces ressources. Les performances d'un programme parallèle dépendent beaucoup de la manière dont les charges sont reparties entre les processeurs. Par exemple, prenons le cas des processeurs identiques avec des charges différentes sur chaque processeur, la

performance globale sera celle du processeur le plus chargé, car les processeurs moins chargés ayant fini doivent attendre ce dernier. Si les vitesses et les temps de calcul sont connus, la répartition de charge peut être décidée de manière statique. Dans le cas contraire, on peut mettre en place une répartition dynamique. Par exemple, une fois que les processeurs ont fini leurs calculs, ils viennent demander de nouvelles charges. Pour que cette répartition de charge puisse être menée à bien, il est nécessaire d'avoir une représentation décrivant le programme.

III-7- La granularité :

La granularité représente le volume de traitement qu'il y a entre deux communications (ou synchronisations). C'est une mesure qualitative du ratio, entre le volume de calcul et le volume des communications.

Une granularité faible signifie que les périodes de calcul sont relativement courtes par rapport aux périodes de communication. Elle facilite la répartition de charge, mais entraîne un surcoût important de communication et laisse peu d'opportunités d'augmenter les performances.

Une granularité forte, à l'inverse, signifie qu'il y a relativement peu de communication en comparaison avec les périodes de calcul. Cela laisse plus d'opportunités d'augmenter les performances mais la répartition de la charge est beaucoup moins évidente à mettre en place.

IV- Exemples de conception de programmes parallèles :

IV-1- Opérations sur les tableaux :

v Description du problème :

Dans cet exemple on prend un tableau à deux dimensions, et on essaye de manipuler ses éléments sachant qu'ils sont indépendants les uns des autres.

Pour commencer on va d'abord écrire un programme qui calcule les éléments du tableau un par un (séquentiellement).

```
Do j = 1,n
  Do i = 1,n
        a(i,j) = fcn(i,j)
   end do
end do
```

v Partitionnement du travail :

Cas1 : Partitionnement statique

Le tableau est partagé en plusieurs portions identiques, chacune sera affectée pour une tâche afin d'être exécutée par un processeur. Dans ce cas, le tableau est divisé par le nombre de processeurs.

Chaque tâche aura la position du départ **(mystart)** et celle de l'arrêt **(myend)** correspondant à la portion du tableau qui lui est affectée comme ceci :

```
do j = mystart, myend
    do i = 1,n
       a(i,j) = fcn(i,j)
    end do
end do
```

Le programme complet utilise deux processus, master et worker :

Le processus master initialise le tableau, envoie les informations au processus worker et à la fin reçoit les résultats.

Le processus worker reçoit les informations correspondantes à la portion du tableau, exécute son code et envoie les résultats au processus master.

```
find out if I am MASTER or WORKER
 if I am MASTER
        initialize the array
        send each WORKER info on part of array it owns
        send each WORKER its portion of initial array
        receive from each WORKER results
else
        receive from MASTER info on part of array I own
        receive from MASTER my portion of initial array
        # calculate my portion of array
        do j = my first column,my last column
                do i = 1,n
                a(i,j) = fcn(i,j)
                end do
        end do
        send MASTER results
endif
```

Figure 2.4 : Partitionnement statique du travail (Opérations sur les tableaux).

- Dans cet exemple on ne trouve pas de dépendances de données puisque les éléments du tableau sont indépendants, et aussi les tâches n'ont pas besoin de se communiquer entre elles.

Cas2 : Partitionnement dynamique

La solution précédente utilise le partage statique du travail, chaque tâche à une quantité fixe de travail à faire, cependant cette solution peut induire à une perte de temps pour les tâches qui s'exécutent plus vite que les autres quant les processeurs utilisés n'ont pas la même vitesse d'exécution.

Pour éviter la perte de temps on utilise un pool de tâche, les processus worker ne savent pas à l'avance quel portion du tableau ou combien de tâches vont exécuter. Les tâches qui s'exécutent plus rapide auront plus de travail à faire.

Le programme utilise deux processus master et worker. Le processus master tient le pool de tâche pour le processus worker, envoie au processus worker une tâche à la demande ensuite collecte les résultats. Le processus worker obtient la tâche du processus master, exécute son code ensuite envoie les résultats pour le processus master.

```
find out if I am MASTER or WORKER
        if I am MASTER
                do until no more jobs
                        send to WORKER next job
                        receive results from WORKER
                end do
                tell WORKER no more jobs
        else
                do until no more jobs
                        receive from MASTER next job
                        calculate array element: a(i,j) = fcn(i,j)
                        send results to MASTER
                end do
        endif
```

Figure 2.5 : Partitionnement dynamique utilisant un pool de tâche.

IV-2- Une équation qui calcule la chaleur :

v Description du problème :

Cette équation décrit le changement de la température à travers le temps, la température est égale à 0 dans les frontières et chaude au milieu.

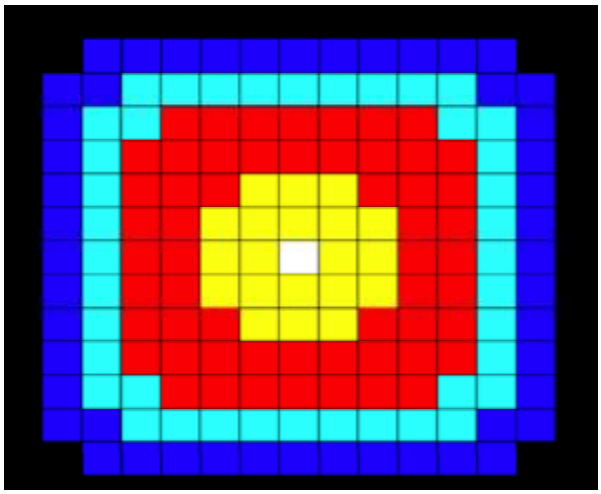


Figure 2.6 : Calcule de la chaleur

Le calcul d'un élément du tableau se fait comme suit :

$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U)$$

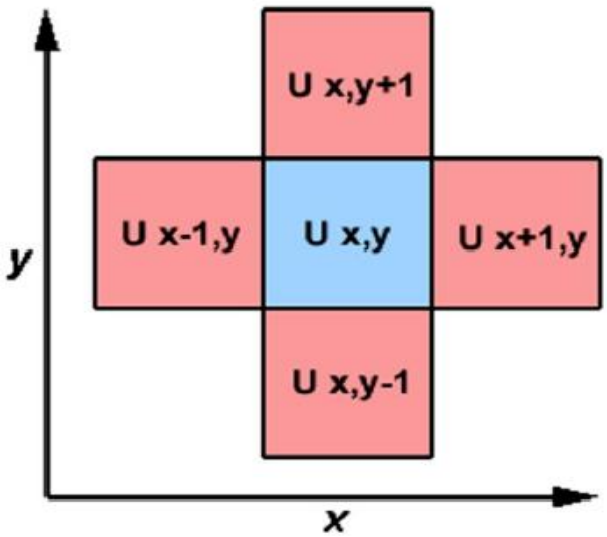


Figure 2.7 : Equation qui calcule la chaleur

Le programme séquentiel correspondant est le suivant :

```
Do iy = 2, ny - 1
        Do ix = 2, nx - 1
                U2 (ix, iy) =
                U1 (ix, iy) +
                Cx * (U1 (ix+1, iy) + U1 (ix-1, iy) - 2* U1 (ix,iy)) +
                Cy * (U1 (ix, iy+1) + U1 (ix, iy-1) - 2* U1 (ix,iy))
        End do
End do
```

v Partitionnement du travail :

Le tableau sera partitionné en sous tableaux qui seront ensuite affectés aux tâches.

v Dépendance de données :

Ce problème nécessite de la communication entre les tâches, car le calcul de la chaleur pour une case donnée du tableau nécessite la valeur de la case voisine.

Dans ce cas on trouve deux sortes de dépendances de données :

Les éléments intérieurs : sont dépendants des éléments dans la même tâche.

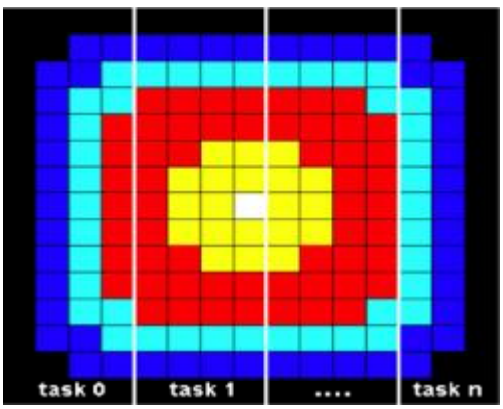
Les éléments de bordure : sont dépendant des éléments qui sont de la même tâche et les éléments de bordure des autres tâches.

v Communication entre les tâches :

Cas1 : Approche avec blocage

Pour les éléments intérieurs, on n'a pas besoin de communication puisque les voisins se trouvent dans la même portion du tableau. Mais pour les éléments de bordure, les tâches doivent communiquer pour fournir les données correspondantes à l'élément à calculer.

Le schéma suivant montre la répartition du tableau entre les différentes tâches :



On définie deux processus **master** et **worker** :

Le processus master envoie les informations initiales pour les processus worker et collecte les résultats.

Le processus worker exécute le code correspondant (calcul de la chaleur), communique si nécessaire avec les autres processus.

Le code suivant montre la solution parallèle du problème :

```
find out if I am MASTER or WORKER

if I am MASTER
        initialize array
        send each WORKER starting info and subarray
        do until all WORKERS converge
                gather from all WORKERS convergence data

                broadcast to all WORKERS convergence signal
        end do

        receive results from each WORKER

else
        receive from MASTER starting info and subarray
        do until solution converged
                update time
                send neighbors my border info
                receive from neighbors their border info
                update my portion of solution array
                determine if my solution has converged
                send MASTER convergence data
                receive from MASTER convergence signal
        end do
send MASTER results
endif
```

Figure 2.8 : Solution parallèle 1 (approche avec blocage).

Cas2 : Approche sans blocage

Dans la solution précédente, nous avons utilisé une communication avec blocage c'est-à-dire les processus nécessitant la communication doivent attendre jusqu'à la disponibilité de l'information pour continuer leur exécution. Les tâches voisines communiquent les données de bordure, ensuite chaque processus met à jour sa portion dans le tableau.

En utilisant une approche de communication sans blocage, on peut réduire le temps d'attente des processus nécessitant la communication. Cette approche permet la continuation de l'exécution en attendant la disponibilité de l'information de la communication.

Chaque tâche doit calculer les éléments intérieurs qui ne nécessitent pas de la communication, pendant qu'elle attend les communications des autres tâches pour calculer les éléments de bordure.

Le code suivant montre la solution parallèle du problème en utilisant une approche de communication sans blocage :

```
find out if I am MASTER or WORKE

if I am MASTER
        initialize array
        send each WORKER starting info and subarray
        do until all WORKERS converge
                gather from all WORKERS convergence data
                broadcast to all WORKERS convergence signal
        end do

        receive results from each WORKER

else
        receive from MASTER starting info and subarray

        do until solution converged
                update time
                non-blocking send neighbors my border info
                non-blocking receive neighbors border info
                update interior of my portion of solution array
                wait for non-blocking communication complete
                update border of my portion of solution array
                determine if my solution has converged
                send MASTER convergence data
                receive from MASTER convergence signal
        end do
        send MASTER results
endif
```

Figure 2.9 : Solution parallèle 2 (approche sans blocage).

V- Conclusion :

Au cours de ce chapitre nous avons étudié le processus de conception d'un programme parallèle, de la première étape à la dernière. Et les exemples présentés à la fin du chapitre nous ont permis de mieux assimiler ce processus en l'appliquant sur des exemples concrets.

Le programmeur trouve souvent des difficultés dans la gestion du parallélisme des problèmes de très grande taille. Pour cela des outils dédiés pour la mise en œuvre des programmes parallèles ont été envisagés.

Dans la partie qui suit, nous allons voir ces différents outils utilisés pour le parallélisme dans les applications desktops.

I- Introduction :

Pour un bon parallélisme dans les applications desktops, les chercheurs ont mis en œuvre plusieurs outils, afin d'aider et de faciliter la tâche aux programmeurs/développeurs pour bien concevoir leurs programmes parallèles à partir des programmes séquentiels déjà existants.

Nous allons ouvrir le chapitre **III** avec un petit aperçu sur les applications desktops, nous passerons ensuite aux différents outils de parallélisme tels que : **Intel TBB** ; **Qt C++** et **ParaTask,** qui sont des bibliothèques parallèles réduisant d'une façon remarquable la durée de développement des applications desktops.

II- Les applications desktops : [PSK, 2008]

Les utilisateurs interagissent avec les applications dans différentes manières, y inclue les tâches concurrentes **(B)**, l'interaction de beaucoup de données **(A),** les I/O**(C)**, l'impression**(D)**, et les dépendances entre tâches **(E)**.

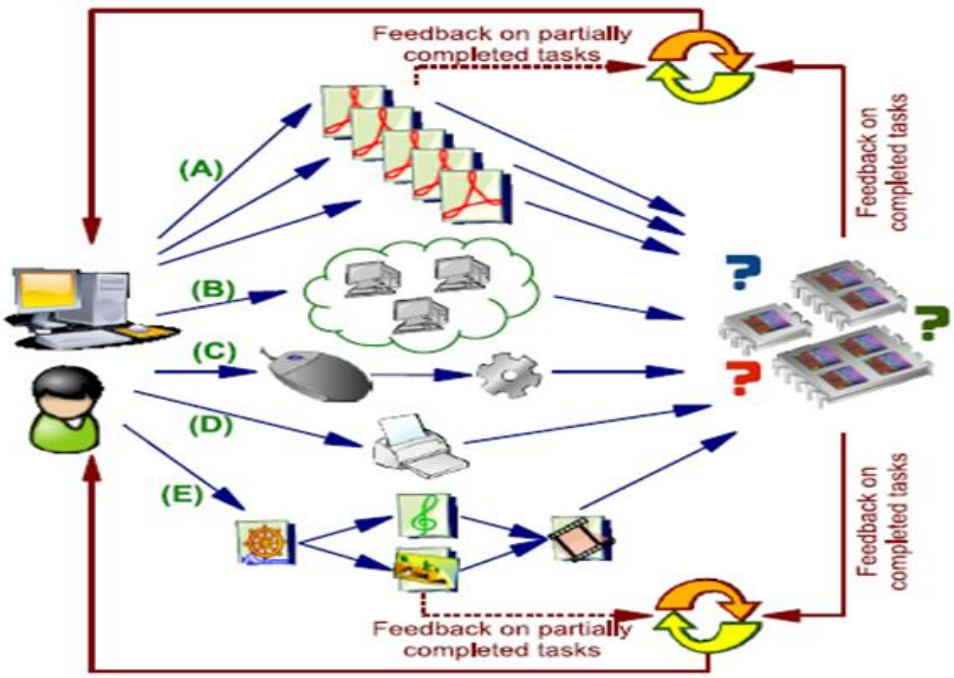


Figure 3.0 : les différents types de tâches dans les applications Desktop.

- **L'anatomie des applications de bureau :**

Avant d'essayer de paralléliser les applications desktops, nous devons comprendre leurs compositions externes et internes.

Les dispositifs externes sont les plus familiers, qui incluent de nombreux composants visuels d'entrée comme les boutons, champs de texte, label… etc, l'organe le plus essentiel dans la structure interne est la boucle d'événement « event loop », qui détecte les événements

(comme le clique de la sourie par exemple), pour les associer à des événements « handler » appropriés **(figure 3.1-2)**.

Le thread GUI appelé aussi « Event Despatch Thread» (EDT), se charge des événements qui viennent des composants externes pour effectuer des traitements internes, aucun autre thread ne peut exécuter cette action, cependant si la boucle d'événement ne traite pas les événements assez rapidement l'application se bloque.

Pour résoudre ce problème on utilise le multithreading, dans des systèmes monoprocesseurs, les threads partagent le processeur dans le temps, le calcul est passé à un autre thread « Thread Helper » **(figure 3.1-3)** et le thread GUI retourne à la boucle d'événement ainsi les deux threads restent actifs, quoique le thread helper exécute le calcul **(figure 3.1-4),** il n y a aucun accès direct aux composants **(figure 3.1-5),** car sauf le thread GUI qui est responsable des composants GUI, ainsi les événements doivent être capturer par le thread GUI **(figure 3.1-7)** et gérer par le thread helper **(figure 3.1-6)** .

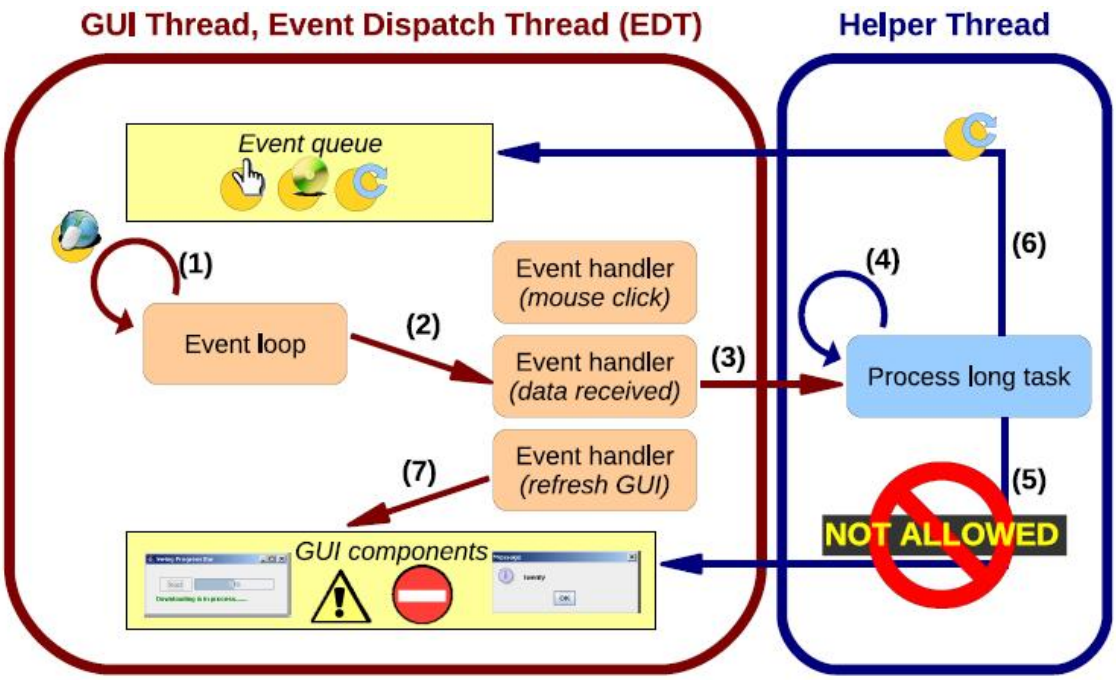


Figure 3.1: Structure du multithreading GUI dans les applications desktops.

III- Parallélisme avec Intel TBB :

Intel crée une librairie qui a pour but d'offrir une interface agréable pour les développeurs en C++ qui souhaitent paralléliser leur code, elle est disponible comme un produit commercial et open source et s'exécute dans des plateformes différentes telles que Windows, Linux et Mac OS.

III-1- Description d'Intel TBB : [Web 05]

Intel TBB (Intel Thread Building Blocks) est une bibliothèque générique C++ développée par la société Intel pour l'écriture des programmes sur des processeurs multicoeurs. La bibliothèque contient des structures de données génériques et des algorithmes qui permettent de manipuler les contenus de ces structures. Grâce à ces structures et algorithmes, elle permet de décharger le développeur de la gestion des processus légers (création, synchronisation ou terminaison).

En plus, elle permet de faire abstraction de l'architecture sous-jacente (le programme ne dépend pas des caractéristiques de l'architecture cible). Un programme TBB est découpé récursivement en plusieurs tâches et celles-ci sont placées dynamiquement sur des ressources disponibles, TBB implémente une stratégie d'ordonnancement par vol de travail (le work-stealing). Dans un programme TBB l'utilisateur spécifie les tâches à exécuter ; une tâche TBB est un ensemble d'instructions indivisibles.

III-2- Caractéristiques de la bibliothèque :

- Permet de créer des applications parallèles fiables, portables et évolutives.
- Contient des modèles de programmation permettant un développement simple et rapide d'applications parallèles stables et capables de s'adapter aux nombre de coeurs disponibles sur un processeur donné.
- Fournit un jeu de composants nécessaires à l'implémentation d'un code parallèle.
- Compatible avec de nombreux environnements et facile à maintenir.

III-3- Intel TBB et le Work-stealing: [WK, MV] [Web 06]

Intel TBB utilise la notion de work-stealing pour distribuer les tâches aux différents threads workers, chaque thread worker maintient les deux extrémités de sa liste des tâches. Quand une tâche est engendrée, celle ci sera mise dans la queue de la liste des tâches d'un thread worker. Quand ce dernier fini l'exécution, il prend une autre tâche de la queue de la liste.

Le point essentiel qu'apporte cette technique est la capacité de rediriger une tâche d'une file locale à une autre file locale en fonction des taux d'occupation sans aucun verrouillage. En d'autre terme, lorsqu'un thread worker constate que sa file est vide, il « vole du travail » à un autre contentant des tâches en attente. Le retrait de la tâche se fait par le haut (tête) de la file concernée, puis elle est réaffectée au contexte de la file couramment analysée.

Pour illustrer la notion work-stealing, imaginons trois tâches dans la file du thread worker **1**, les tâches **3** et **4** sont en attente alors que la **5** est en cours d'exécution **(figure à gauche)**.Comme on le constate, il n'y plus rien à dépiler dans la file globale, mais pour

améliorer les performances de l'ensemble, la tâche **3** est déplacée, ou "volée" de sa file d'origine, pour être distribuée au thread de travail disponible **(figure à droite)**.

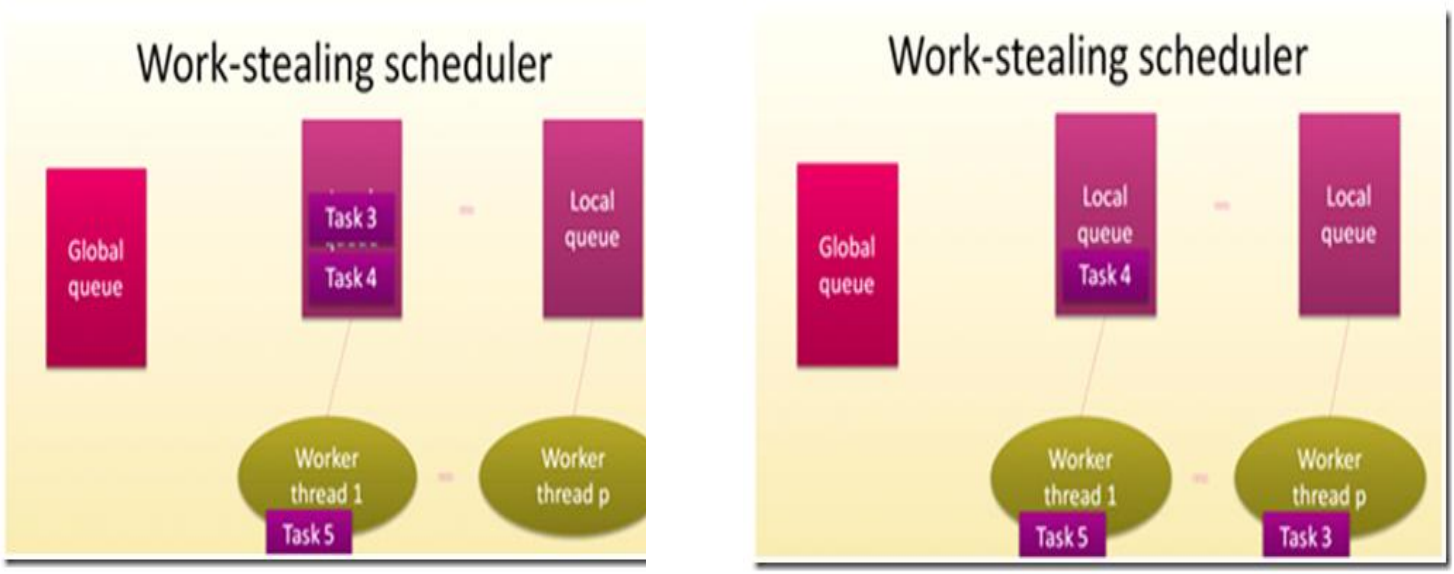


Figure 3.2: La notion work stealing

III-4- Les composants d'Intel TBB : [WK, MV]

La figure qui suit, illustre les différents composants offerts par intel TBB; pour aider le développeur dans la construction de ses programmes parallèles.

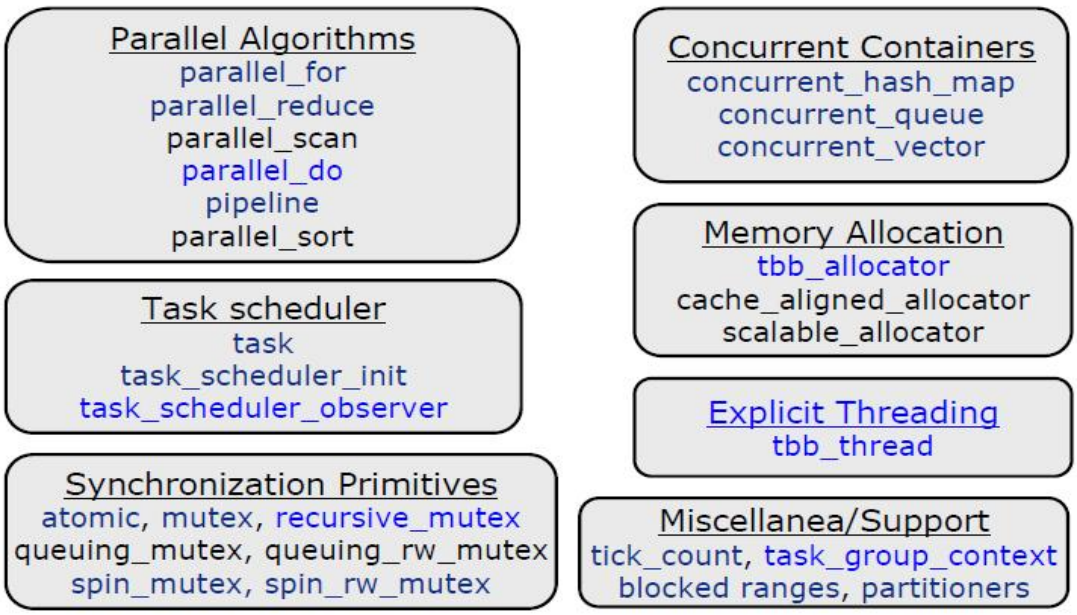


Figure 3.3 : Composants d'Intel TBB.

III-4-1- Utilisation des tâches :

Intel TBB est basée sur le concept de tâches, elle fournie deux classes pour utiliser ces tâches: **task-group** et **task**. La classe **task-group** définie une méthode « **run »,** dont chaque appel crée une tâche exécutant ce qui est passé en argument.

Pour permettre au programmeur de contrôler le comportement bas niveau tels que : l'affectation des tâches pour les threads, et la gestion des exceptions, on utilise la classe **task**. Cette dernière a aussi la possibilité d'exprimer les dépendances complexes entre les différentes tâches crées.

III-4-2- Les algorithmes parallèles génériques :

Intel TBB fournie un ensemble de packages d'algorithmes génériques, construit pour faciliter l'utilisation des tâches.

L'utilisateur spécifie une collection d'objet et le traitement associé, la bibliothèque crée les tâches à travers cette collection par des subdivisions récursives. En premier, elle sera divisée en deux taches, ensuite c'est à ces dernières ; de décider si elles vont encore êtres subdiviser ou pas; la collection en deux autres tâches.

IntelTBB fournie trois politiques de répartition, pour contrôler la terminaison de cette récursivité :

- **Simple-partitioner** : divise le rang s'il est plus grand que le seuil définit par l'utilisateur.

- **Auto-partitioner** : par défaut, s'il n y a pas de tâche à prendre des autre listes (work-stealing), divise le rang en p*4 sous rang (tel que : p est le nombre de thread hardware), cependant si une tâche est prise (volée), le thread qui prend cette tâche, la divise en p*4 sous tâches

- **Affinity-partitioner** : cette politique garde l'historique des threads qui ont exécutés un sous rand donné, et essayer de maintenir cette distribution dans les futurs exécutions.

Parmi ces algorithmes parallèles, on cite : **Parallel-for** qui est un algorithme montrant comment repartitionner les tâches. Et les algorithmes **Parallel-reduce** et **Parallel-scan** qui appliquent un corps d'instruction à une collection d'éléments d'une façon concurrente.

III-4-3- Les containers concurrents:

Utilisés par les threads, dans les applications desktop parallèle, pour la synchronisation et la communication, Intel TBB fournit les conteneurs concurrents, y compris vecteur, file d'attente, et hachage. Ils utilisent des éléments de verrouillage pour éviter les conflits d'accès simultanés par plusieurs threads. Lorsque vous accédez à un conteneur standard à partir de plusieurs threads, il est nécessaire de protéger ces accès en écriture à exclusion mutuelle.

Intel TBB offre trois types de containers concurrents:

v Les containers concurrents associatifs :

Appliquent un accès concurrent à une table de hachage.

« **Concurrent_hash_ map » :** Est une classe qui assure l'insertion, le verrouillage et la modification dans la même instance d'une table. Utilisant un pointeur appelé accessor ; qui se pointe sur la table de hachage, prend un verrou provisoire jusqu'a ce qu'il soit détruit. Si la table contient la paire (key, value) correspondant à la clé d'entrée, les méthodes **insert()** et **find()** la retournent pour l'accessor, et la méthode **erase()** détruit cette donnée de la table puis libère le verrou.

v Les vecteurs concurrents:

« Concurrent_vector » : Cette classe est un tableau d'éléments qui permet l'accès concurrent. Les threads peuvent ajouter des éléments au tableau, sans connaitre les indices, ceci a deux conséquences importantes: les méthodes **erase()** et **insert()** ne peuvent pas être utilisées, et un élément ne peut pas être allouer plus d'une seul fois.

L'exemple (le code) suivant montre comment insérer un chaine de caractère à un tableau partagé en utilisant la méthode **std:copy ().**

```
Void Append ( concurrent_vector<char>& vec, const char* str ) {
size_t n = strlen(str)+1;
std::copy ( str, str+n, vec.grow_by(n) );

}
```

v Les files concurrentes:

« Concurrent_queue » : Cette classe est une structure de donnée de type FIFO, qui permet aux threads d'insérer et d'ajouter de nouveaux items, la taille de la file n'est pas fixée. Ainsi destinée pour les machines à mémoire limitée. Cette classe ne comporte pas les méthodes de verrouillage ou de blocage.

Les opérations fondamentales de cette classe sont: **push()** et **try_pop(),** la première permet l'insertion dans la file, et puisque la taille de la file n'est pas fixée elle renvoie toujours le résultat vraie (si l'espace mémoire est suffisant). La seconde récupère un item de la file s'il est disponible.

III-4-4- Les primitives de synchronisation :

En plus des algorithmes parallèles de haut niveau, Intel TBB fournie d'autres composants de bas niveau tels que les verrous, et les opérations atomiques.

v Les verrous :

Les verrous dans IntelTBB sont faciles à utiliser et permettent une programmation générique, et utilisent la méthode **scoped_lock()** qui implémente les méthodes **acquérir()** et **restituer().**

La classe la plus simple est **spin-mutex,** le thread se boucle en attendant que le verrou soit disponible (l'instruction **spin_mutex::scoped_lock ;** essaye d'acquérir le verrou, si le verrou n'est pas disponible le thread se boucle, sinon elle l'acquit . À la fin le destructeur restitue le verrou).

v Les opérations atomiques: [Web07]

Intel TBB fournie cinq opérations fondamentales décrites dans le tableau suivant:

Opérations	**Descriptions**
=x	Lire x.
x=	Ecriture de x et retourne sa valeur
x.fetch_and_store(y)	x=y, et retourne la nouvelle valeur de x.
x.fetch_and_add(y)	x+=y, et retourne la nouvelle valeur de x.
x.compare_and_swap(y,z)	Si x=z alors x=y, et retourne la nouvelle valeur de x.

IV- Parallélisme avec Qt C++ :

IV-1- Présentation de Qt C++: [Web 06]

Qt est un framework multi-plateforme (s'adapte à tous les OS(s)), qui permet de créer des GUI « **G**raphical **U**ser **I**nterface, soit "Interface utilisateur graphique" », désignant tout ce qu'on appelle grossièrement "Programmation avec des fenêtres".

Qt est écrit en C++ et est fait pour être utilisé à la base en C++, mais il est aujourd'hui possible de l'utiliser dans d'autres langages comme Java, Python, etc.

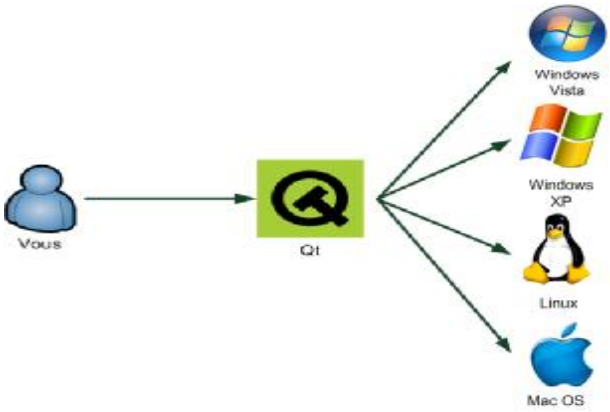


Figure 3.4 : Qt multi-plateforme.

Qt est basé sur le mécanisme de **SIGNAL** et de **SLOT**.

- **Un signal :** est un message envoyé par un widget (élément d'une interface graphique) lorsqu'un évènement se produit (un cliqué sur un bouton par exemple).

- **Un slot :** est la fonction qui est appelée lorsqu'un évènement s'est produit. On dit que le signal appelle le slot. Concrètement, un slot est une méthode d'une classe. **Exemple :** le slot quit() de la classe QApplication, qui provoque l'arrêt du programme.

IV-2- Le concept de tâche : [GS, parallel]

Qt est utilisé pour les environnements desktop (KDE), et implémente le concept de tâche. On peut produire le programme séquentiel équivalent, en supprimant les mots clés liés au parallélisme ; ainsi il peut être exécuté avec n'importe quel compilateur standard C++, ceci parce que le parallélisme est introduit sans modifier la structure du programme original.

IV-2-1- Déclaration et définition d'une tâche :

La déclaration d'une tâche se fait comme n'importe quelle déclaration typique d'une fonction de C++, elle se fait avec le mot clé « TASK » :

TASK void task1 (String filename);

Cet exemple prend en paramètre une image et réduit sa taille. La définition reste comme une fonction C++:

void task1(String filename) {/* user code */}

IV-2-2- Invocation d'une tâche :

L'invocation d'une tâche se fait aussi comme invoquer une tâche en C++, mais avec deux différences : premièrement contrairement à un appel de fonction, l'exécution d'une tâche se fait d'une manière asynchrone avec l'appelant. Deuxièmement, l'invocation retourne un objet TaskID.

Le code ci-dessous représente une boucle à travers une collection d'images, et invoque une tâche pour chaque image :

```
List images = ...;
    For (int i = 0; i < images.size (); i++)
        {TaskID *id = task1 (images.at(i);}
```

IV-2-3- Spécification des dépendances entre les tâches :

Dans le parallélisme des tâches, il est possible que certaines tâches ne puissent pas continuer leurs exécutions jusqu'à ce que les autres tâches terminent leurs exécutions, Par exemple, la figure ci après montre quatre tâches; les tâches **2** et **3** doivent attendre la tâche **1**, tandis que la tâche **4** doit attendre les tâches **2** et **3**.

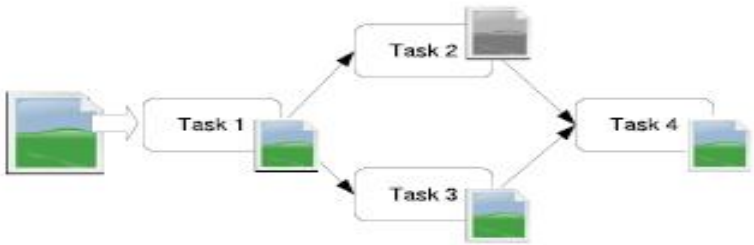


Figure 3.5 : Dépendance entre les tâches.

La spécification se fait en déclarant les dépendances au moment où la tâche est appelée:

```
For (int i=0; i < images.size(); i++){
    TaskID *id1 = task1 (images.at (i));
    TaskID *id2 = task2 (images.at (i)) dependsOn(id1);
    TaskID *id3 = task3 (images.at (i)) dependsOn(id1);
    TaskID *id4 = task4 (images.at (i),
    images.at (i)) dependsOn (id2, id3); }
```

Les tâches **2**, **3** et **4** ne peuvent pas continuer l'exécution jusqu'à ce que les tâches spécifiées par le mot clé « DependsOn » vont terminer leurs exécutions. De ce fait le programmeur n'a pas besoin de mécanismes de synchronisation pour gérer ces dépendances.

IV-2-4- Terminaison et récupération des résultats d'une tâche :

Lors de l'appel d'une tâche asynchrone, il est important de savoir quand celle-ci se termine. Que ce soit une tâche qui renvoie de résultats ou qui ne renvoie pas de résultats, celle-ci est invoquée de la même manière:

```
TaskID *idReturn = taskWithReturn();
TaskID *idNoReturn = taskWithNoReturn();
```

Ø Approche avec blocage :

Pour les tâches renvoyant des valeurs, le résultat est accessible par la méthode **result()** sur le TaskID. Si la tâche est déjà accomplie, le résultat est retourné immédiatement par **result().** Autrement, le programme appelant se bloque jusqu'à la fin de la tâche.

```
int answer = idReturn->result<int>();
```

Ø Approche sans blocage:

L'approche avec blocage pose un problème pour les programmes interactifs qui se basent sur les événements, pour régler ce problème, le concept de tâche fournie une approche sans blocage, quand la tâche est invoquée, une liste de slots est spécifiée en utilisant la clause « **notify »**, ces slots sont invoqués automatiquement quand la tâche est complétée.

```
TaskID *idReturn = taskWithReturn ()
Notify (mySlot(TaskID*));
```

IV-2-5- Les différents types de tâche :

Ø Les tâches one-off :

Les tâches introduites jusqu'à présent sont toutes des tâches OneOff. Lorsqu'elles sont invoquées, une seule est empilée pour être exécuté par un thread worker. La sémantique d'une tâche one-off, est qu'elle s'exécute à partir du début jusqu'à la fin sur un thread worker, quand elle se termine, une autre tâche est planifiée sur le même thread worker.

Ø Les tâches interactives :

Tâches avec une longue durée d'exécution et qui se bloquent fréquemment, et empêchent les threads d'exécuter d'autres tâches. Pour cette raison, les tâches interactives sont prévues. Elles sont déclarées comme les tâches one-off, mais sont invoqués avec le mot-clé **« interactive »:**

```
TaskID *id = waitForInput() interactive;
```

Ces tâches s'exécutent sur un thread interactif séparé (pas en file d'attente sur un thread worker) et s'exécutent aussi sans l'attente des threads workers libres.

Ø Les multitâches :

Il y a une différence entre l'invocation d'une multitâche une seul fois et l'invocation d'une tâche one-off plusieurs fois. Premièrement, une multitâche fournie un meilleur contrôle, deuxièmement, les sous tâches de la multitâche sont organisées en différents threads worker dans un round robin (une technique qui utilise une liste circulaire pour allouer les tâches).

L'avantage majeur est qu'une multitâche sait ce qui se passe dans le groupe, et ceci peut être utilisé pour le partage efficace des tâches, de plus de globalID ; les concepts suivant sont utilisés pour les multitâches :

Multi-task size : Représente le nombre des sous tâches dans une multitâche.
Relative ID : Représente l'identifiant de la sous tâche avec le résultat de la sous tâche dans la même multitâche (débute à 0 et finis dans le multitask size-1).

```
TASK (N) void multiTask() {
        int myPos = Task::relativeID();
        int s = Task::multiTaskSize();
        If (myPos == 0)
        Print ("Multi-task has"+s+"sub-tasks");
        Print ("Hello from sub-task"+myPos);
                }
```

IV-3- Implémentation:

IV-3-1- Planification des tâches :

La **figure 3.6** illustre l'invocation d'une tâche, quant cette dernière est invoquée, elle est empilée dans une file **(étape1).** La tâche est placée sur le pool de tâches **(étape2)** et une nouvelle valeur TaskID est retournée **(étape3)**, Cela permet à l'appelant de continuer son exécution **(étape4),** alors que la tâche s'exécute d'une façon asynchrone sur un thread worker **(étape5).**

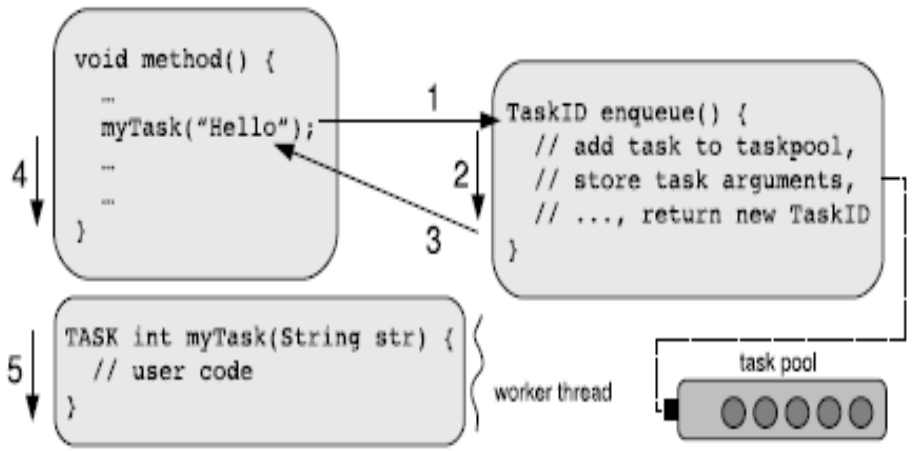


Figure 3.6 : Mise en file d'une tâche.

IV-3-2- Système d'exécution:

Le système d'exécution illustré dans la **figure 3.7** est responsable de la gestion des tâches.

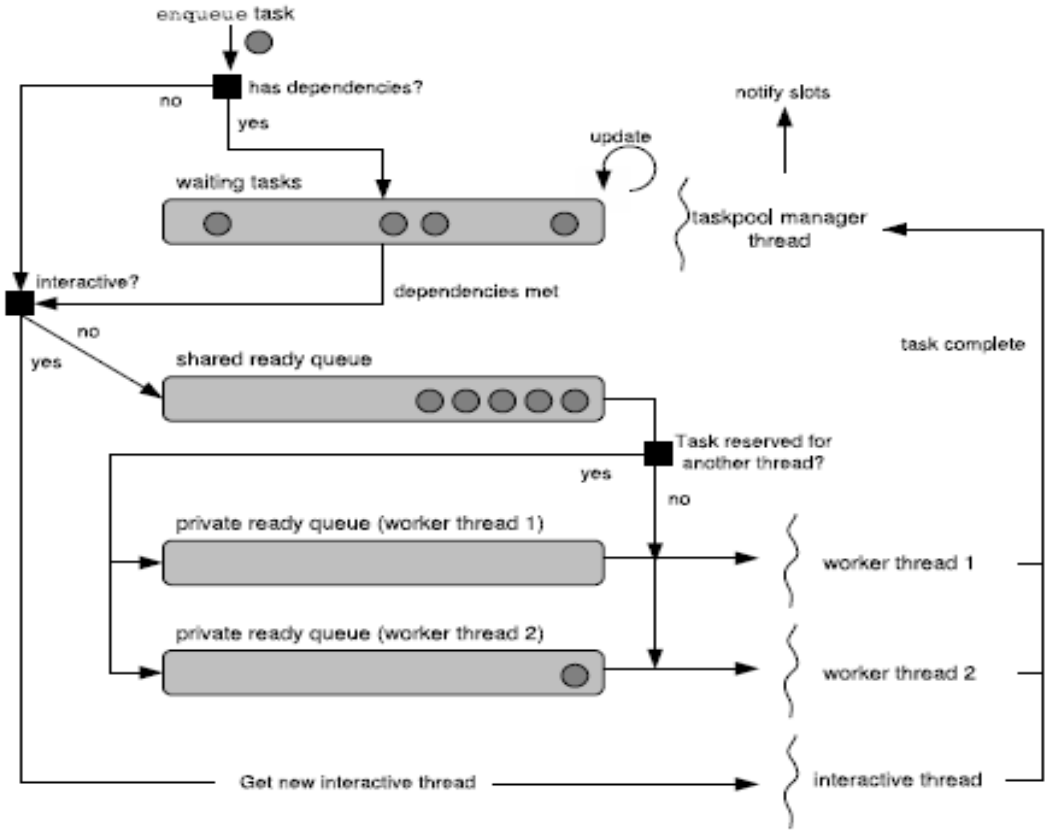


Figure 3.7 : Implémentation du système d'exécution.

Ø Mise en file d'une tâche :

Quand une tâche est en file d'attente, des connexions sont établies pour notifier les slots enregistrés. Si la tâche dépend des autres tâches, elle sera stockée avec les tâches en attente. Sinon, la tâche est prête à s'exécuter: une tâche non interactive est stockée sur la file d'attente partagée (et tous les threads workers endormis sont réveillés) alors qu'une tâche interactive s'exécute sur un nouveau thread interactif.

Ø Thread worker est prêt :

Toutes les tâches, à l'exception des tâches interactives, sont exécutées par des threads workers. Premièrement, le thread worker exécute toutes les tâches dans sa file d'attente privé. Deuxièmement, une tâche est prise dans la file d'attente partagée prête. Si la tâche est réservée à un autre thread (c-à-d dans le cas de multi-tâches), elle est mise en attente dans la file d'attente privée de ce dernier. Sinon, elle est exécutée. Troisièmement, le thread worker dort s'il ne reste aucune tâche dans ses files d'attente privée ou partagée.

Ø Achèvement d'une tâche :

Lorsqu'une tâche termine son exécution, le pool de tâches est responsable de la mise à jour des tâches en attente et de notification des slots. Si une tâche en attente a maintenant ses dépendances satisfaites, elle est soit en file d'attente sur la file d'attente partagée, soit exécutée sur un nouveau thread interactif.

V- Parallélisme avec ParallelTask: [NG, OS]

V-1- Définition de ParallelTask :

Le concept de ParallelTask connu sous le nom de ParaTask a été révisé, amélioré et étendu. Il a été utilisée avec succès pour les grosses applications (des applications Web en utilisant l'API, et de traitement de photos diverses…).

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

V-2- Le concept de tâche :

V-2-1- Définition et invocation des tâches :

Paratask encapsule les tâches au niveau des méthodes, la définition se fait avec le mot clés « TASK », voici une définition d'une tâche qui détecte les contours d'une image :

```
TASK public Image edgeDetectTask (Image i)
{// detect the edges}
```

Cet événement handler montre l'invocation de cette tâche :

```
Public void actionPerformed (ActionEvent e) {
... For (Image image: selectedImages) {
TaskID<Image> result = edgeDetectTask(image); ... } }
```

Invoquer une tâche parallèle avec ParaTask, se fait de la même manière qu'invoquer une méthode standard de java, à l'exception que la première s'exécute en parallèle avec le programme appelant. L'invocation retourne un objet « **TaskID** ».

La méthode **actionPerformed** est un évenement hendler, c'est le thread GUI qui la traite.

Dans une implémentation séquentielle, le thread GUI détecte les contours entiers de l'image, l'application se bloque en ce moment. Cependant dans le mode parallèle, le thread GUI place les tâches dans une file puis retourne à la boucle d'événement (event loop), ensuite les tâches vont s'exécutées par un groupe de threads.

Paratask crée et gère automatiquement un nombre idéal de threads pour garder chaque unité de calcul occupé, donc invoquer plusieurs tâches est plus efficace que créer un thread pour chaque traitement. Maintenant que la tâche est chargée pour un autre thread, le thread GUI doit savoir le résultat d'exécution de la tâche.

```
Image i = result.getResult();
```

Si la tâche est déjà complétée, il n y aura aucun problème, le résultat est disponible sinon, le thread qui a invoqué cette tâche se bloque jusqu'à ce que la tâche soit terminée. Ce concept est inaccessible par le thread GUI, donc on aura besoin d'un moyen pour que le thread GUI soit au courant de l'accomplissement de la tâche :

Voici une solution qui propose une approche sans blocage :

```
Public void actionPerformed (ActionEvent e) {
...
For (Image image: selectedImages) {
TaskID<Image> result = edgeDetectTask(image)
notify (updateGUI(TaskID));
...                                    }
}
Public void updateGUI(TaskID<Image> id) {
Image thumbnail = id.getResult();
// display thumbnail, update progress bar...
}
```

Le thread GUI retourne à la boucle d'événement (event loop), puis il se rend compte de l'accomplissement de la tâche pour mettre à jour l'application en utilisant la clause **notify.**

V-2-2- Dépendances entre les tâches :

Dans les programmes séquentiels, les contraintes de commande sont obéi naturellement, puisque des segments de code sont exécutées une à la fois dans l'ordre spécifié. Regardons le graphe de dépendance **(figure 3.7)**. Cela exige que **compute2** et **compute3** ne commencent pas avant **compute1** est terminée, et même **compute4** doit attendre à la fois pour **compute2** et **compute3**.

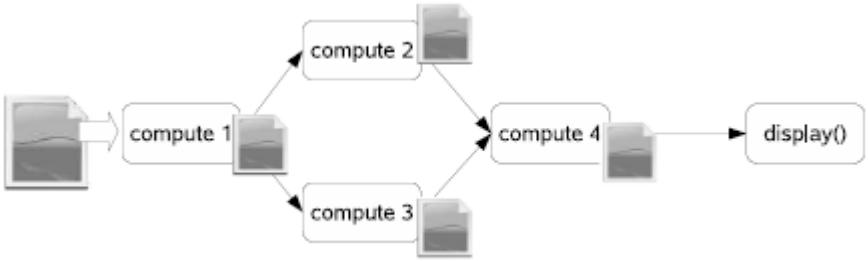


Figure 3.8 : Quatre tâches avec dépendances entre elles.

ParaTask a la possibilité de gérer ces dépendances en les indiquant avec le mot clé « dependsOn » comme suit:

```
Public void actionPerformed() {
TaskID<File> id1 = compute1 (“myimage.jpg”);
TaskID<File> id2 = compute2 (“myimage1.jpg”)
dependsOn(id1);
TaskID<File> id3 = compute3 (“myimage1.jpg”)
dependsOn(id1);
TaskID<File> id4 = compute4 (“myimage2.jpg”,
“myimage3.jpg”) dependsOn (id2, id3);
}
```

Les tâches de 2, 3 et 4 ne peuvent pas continuer jusqu'à ce que les tâches spécifiées par « DependsOn » soient terminées.

V-2-3- Les différents types de tâche :

Ø Les tâches interactives :

Considérons une tâche qui ne réalise qu’un simple calcul ou interagisse avec l’extérieur par exemple une recherche sur internet, dans ce cas, assigner cette tâche pour un thread worker est indésirable s’il y en a d’autres calculs intensifs qui peuvent se faire avec ce thread worker.

Paratask définie les tâches interactives comme suit :

```
INTERACTIVE_TASK public List<Image> searchTask(String query)
{
// perform internet search
}
```

paraTask traite les tâches interactives comme les tâches standards, la différence ; c’est que la tâche interactive ne sera pas empilée dans la file du thread worker.

Ø Les multitâches :

Les multitâches en ParaTask permettent de déterminer la position d'une sous tâche dans le groupe (ligne 2 et 4) et une barrière de synchronisation entre les sous tâches (ligne 12).On définie une multitâche comme suit :

```
1 TASK (*) public void multiTask(ParIterator<File> pi) {
2 int myPos = CurrentTask.relativeID();
3 print ("Hello from sub-task"+myPos);
4 int numTasks = CurrentTask.multiTaskSize();
5 if (myPos == 0)
6 print ("Multi-task has "+numTasks+" sub-tasks.");
7...
8 while (pi.hasNext() ) {
9 process (pi.next() );
10}
11...
12 CurrentTask.barrier();
13}
```

VI- Conclusion :

Les chercheurs ont définis des outils pour le parallélisme dans les applications desktops, en proposant des bibliothèques supplémentaires et des extensions des langages de programmation les plus populaires comme JAVA et C++.

Ce chapitre propose un concept de tâche pour les programmes orientés objet sur les systèmes à mémoire partagée. Ce concept permet une restructuration d'un programme sans modifier son comportement. Cependant un code parallèle très similaire à la version séquentielle est produit. Pour le parallélisme de tâche plus complexe qui nécessite des dépendances, le programmeur est pourvu d'un mécanisme intuitif à utiliser.

I- Introduction :

Nous commençons ce chapitre avec quelques exemples pour mieux comprendre l'outil ParaTask de java, dans sa déclaration et invocation des tâches ; qu'elles soient multitâches, tâches one-off ou tâches interactives.

Nous passerons par la suite à présenter notre future application et cela en décrivant tout d'abord l'environnement et les outils de développement utilisés, on montrera par la suite l'interface d'exécution séquentielle pour l'extraction des contours d'une image se trouvant sur disque, puis l'interface d'exécution parallèle (répartition en bloc et répartition cyclique de données) de la même image, et pour terminer ; nous allons tracer un tableau qui permet de calculer l'accélération et l'efficacité de la programmation parallèle par rapport à la séquentielle.

II- Exemples : [Belkacem 2011]

Exemples 1: **Invocation des tâches:**

Dans cet exemple nous allons expliquer comment invoquer une tâche one-off, une multitâche et une tâche interactive, le programme consiste à afficher une chaine de caractère à travers la méthode hello ().

```
Public static void hello (String name) {
    System.out.println ("Hello from "+name);
}
```

Les trois méthodes suivantes représentent respectivement la définition d'une tâche one-off, multitâche et tâche interactive.

```
TASK public static void task_hello() {
    hello ("Task");
}

TASK (*) public static void multi_hello() {
    hello ("Multi-Task [subtask "+CurrentTask.relativeID()+"]");
}

INTERACTIVE_TASK public static void interactive_hello() {
    hello ("Interactive Task");
}
```

On remarque que les trois méthodes réutilisent le code séquentiel en appelant la méthode hello (), aussi que les trois méthodes sont presque identiques sauf dans quelques différences dans la déclaration de la tâche.

Maintenant que nous avons définie les tâches, on passe à l'invocation. On note que l'affichage des numéros est pour plus de visibilité.

```
System.out.println("(1)");
hello ("Sequential");
System.out.println ("(2)");
TaskID id1 = task_hello ();
System.out.println ("(3)");
TaskID id2 = multi_hello ();
System.out.println ("(4)");
TaskID id3 = interactive_hello ();
System.out.println ("(5)");
```

On rappelle que hello() est la méthode séquentielle, les trois autres méthodes qui sont déclarées par le mot clés TASK retournent un objet TaskID qui correspond à l'invocation de cette tâche, et puisque les tâches s'exécutent d'une façon asynchrone avec le programme appelant on a besoin de synchroniser quand les tâche terminent leurs exécution, une approche est de placer les identificateurs des tâches (TaskID) dans un groupe et attendre jusqu'a ce que l'exécution des tâches qui se trouvent dans ce groupe soit terminé.

```
TaskIDGroup g = new TaskIDGroup (3);
g.add (id1);
g.add (id2);
g.add (id3);
System.out.println ("** Going to wait for the tasks...");
g.waitTillFinished ();
System.out.println ("** Done! All tasks have now completed.");
```

Le paragraphe suivant montre un exemple d'exécution:

```
(1)
Hello from Sequential
(2)
(3)
Hello from Task
(4)
(5)
** Going to wait for the tasks to execute...
```

Hello from Interactive Task

Hello from Multi-Task [subtask 0]

Hello from Multi-Task [subtask 1]

** Done! All tasks have now completed.

Les tâches s'exécutent d'une façon asynchrone avec le programme appelant, les instructions qui suivent l'invocation des tâches s'exécutent au même temps que les tâches.

Exemple 2 : **La différence entre les types de tâches**

Dans cet exemple on expliquera la différence entre les tâches one-off, multitâches et tâches interactive, pour cela on tente d'accéder au web pour calculer le nombre d'accès à un ensemble de sites web donné en entrée. Cependant on aura besoin d'une connexion réseau pour exécuter ce programme.

La méthode qui effectue ce traitement est donnée comme suit :

```
Public static void webAccess(String address) {
…
// Calculer le nombre d'accée.
...
// Afficher le temps de traitement.
...
}
```

Ensuite on crée une collection de sites qu'on veut traiter comme suit:

```
ConcurrentLinkedQueue<String> urls = new ConcurrentLinkedQueue<String>();
urls.add ("http://en.wikipedia.org/wiki/New_Zealand");
...
urls.add ("http://en.wikipedia.org/wiki/Tongariro_National_Park");
```

Maintenant on passe à la définition des tâches, mais on commence d'abord par la méthode séquentielle.

a- La méthode séquentielle:

```
For (String s: urls) {
webAccess (s);
}
System.out.println("Total time: "+time+" seconds.");
```

Un exemple d'exécution de la méthode séquentiel est donné comme suit:

830 links for 'New_Zealand' [11.921 seconds, thread 1]

509 links for 'Auckland' [0.415 seconds, thread 1]

202 links for 'Rotorua' [0.209 seconds, thread 1]

127 links for 'Milford_Sound' [0.21 seconds, thread 1]

524 links for 'Christchurch' [0.613 seconds, thread 1]

467 links for 'Dunedin' [1.971 seconds, thread 1]

152 links for 'Queenstown,_New_Zealand' [0.402 seconds, thread 1]

149 links for 'Tongariro_National_Park' [0.407 seconds, thread 1]

Total time: 16.159 seconds.

Le temps de traitement n'est pas important puisque l'exécution dépend du réseau, ce qui est important est la relation entre les temps individuels et le temps total. Dans ce cas tous les traitements sont exécutés par un seul thread (thread1), et le temps total est la somme des temps individuel de chaque traitement, ceci signifie que l'application est bloqué pendant 16 secondes.

b- Tâches one-off:

La première étape consiste à définir la tâche, et pour la réutilisabilité du code on va encapsuler dedans la méthode séquentielle webAccess():

```
TASK public static void webAccessTask(String address) {

webAccess(address);

}
```

Et après l'invocation des tâches, on ajoute les identificateurs des tâches (TaskID) dans un seul groupe pour synchroniser à la fin.

```
TaskIDGroup g = new TaskIDGroup(urls.size());

for(String s: urls) {

TaskID id = webAccessTask(s);

g.add(id);

}

g.waitTillFinished();
```

Un exemple d'exécution est donné comme suit:

509 links for 'Auckland' [1.682 seconds, thread 7]

202 links for 'Rotorua' [0.415 seconds, thread 7]

830 links for 'New_Zealand' [3.106 seconds, thread 8]

127 links for 'Milford_Sound' [0.21 seconds, thread 7]

524 links for 'Christchurch' [1.804 seconds, thread 8]

152 links for 'Queenstown,_New_Zealand' [2.024 seconds, thread 8]

467 links for 'Dunedin' [1.569 seconds, thread 7]

149 links for 'Tongariro_National_Park' [0.203 seconds, thread 7]

Total time: 6.934 seconds.

Encore le temps individuel n'est pas important. On s'intéresse à la façon dont les threads worker (thread 7 et thread 8) ont exécuté les tâches, le temps total est le maximum des sommes des temps individuels de chaque thread, dans cet exemple le temps total est la somme des temps individuels du thread 8, et prend 6.934 secondes.

Pour la pluparts des traitements, les tâches One-off représente la bonne solution, cependant dans cette application particulière, les tâches One-off ne représentent pas la meilleur solution puisque les tâches interactives vont encore permettre de réduire le temps d'exécution.

c- Multitâches:

Les multitâches sont des tâches One-off à l'exception qu'elles sont exécutées plusieurs fois, ainsi partagées par un nombre fixe de threads workers.

La définition de la multitâche est donnée comme suit:

```
TASK(*) public static void webAccessMulti(ConcurrentLinkedQueue<String>
queue) {
String s = null;
while ((s = queue.poll()) != null)
webAccess(s);
}
```

La différence ici avec la tâche One-off est que toute la collection est passée à la multitâche, chaque sous tâche de la multitâche prend une adresse de la collection d'adresse.

L'avantage des multitâche est que les sous tâches sont placées automatiquement dans un groupe, et sachent ce qui ce passe dans ce dernier.

On passe maintenant à l'invocation de la tâche:

```
TaskIDGroup g = webAccessMulti(urls);
g.waitTillFinished();
System.out.println("Total time: "+time+" seconds.");
```

Dans cet exemple, l'exécution de la multitâche est la même que la tâche One-off, ceci est dû en fait à l'utilisation d'un nombre fixe de threads workers, cependant les multitâche simplifie l'écriture des codes parallèles puisque TaskIDGroup qui contient les TaskIDs est retourné automatiquement.

d- Tâches interactives:

Quoique les tâches One-off sont les plus utilisées dans ParaTask, elles ne présentent pas la bonne solution pour cet exemple, puisque la méthodes webAccess() ne fait pas un calcul compliqué et le thread worker passe son temps dans l'attente du réseau. C'est ici que les tâches interactives sont les plus appropriées: les calculs qui impliquent les entrés/sorties externes.

```
INTERACTIVE_TASK public static void webAccessInteractiveTask(String address)
{
webAccess(address);
}
```

On invoque les tâches interactives de la même manière que les tâches One-off, seulement les tâches ne sont pas partagées par un pool de threads existant, chaque tâche est affectée à sont propre thread worker, et puisque le calcul n'est pas compliqué on préfère exécuter toutes les tâches d'une façon concurrente, puisque les threads workers se bloquent en attente des entrés/sorties.

```
127 links for 'Milford_Sound' [1.069 seconds, thread 13]
149 links for 'Tongariro_National_Park' [1.26 seconds, thread 17]
152 links for 'Queenstown,_New_Zealand' [1.263 seconds, thread 16]
202 links for 'Rotorua' [1.437 seconds, thread 11]
524 links for 'Christchurch' [1.694 seconds, thread 14]
467 links for 'Dunedin' [1.734 seconds, thread 15]
509 links for 'Auckland' [1.917 seconds, thread 10]
830 links for 'New_Zealand' [2.134 seconds, thread 9]
Total time: 2.16 seconds.
```

On note que les tâches sont exécutées par des threads différents, donc le temps total est le temps de la tâche la plus lente.

III-Environnement de développement :

III-1- Le matériel utilisé :

Durant la réalisation de notre application, nous avons utilisé une machine ayant les caractéristiques suivantes :

- Un microprocesseur Intel Core 2 duo.
- RAM DDR de 2 GO.

- Disque dure de 320 GO.

Nous avons développé notre application sous un système d'exploitation Microsoft Windows 7.

III-2-Présentation de l'environnement :

III-2-1- Le langage de programmation utilisé : [JAVA, EYROLLES]

Langage JAVA

Mis au point par Sun Microsystems, Java est un langage de programmation utilisé dans de nombreux domaines. Son succès est dû à un ensemble de caractéristiques dont voici un aperçu:

Ø **Java est indépendant de toute plate-forme :** Une application en java fonctionne sur n'importe quel environnement (Unix, Windows, …) disposant d'une JVM (Java Virtual Machine), en français : la machine virtuelle java.

Ø **Java est extensible à l'infini :** Idéalement, toutes les catégories d'objets (appelées classes) existantes en java sont définies par extension d'autres classes, en partant de la classe de base la plus générale : la classe Object. Pour étendre le langage il suffit donc de développer de nouvelles classes.

Ø **Java est un langage de haute sécurité :** Java a été développé dans un souci de sécurité maximale. L'idée maitresse est qu'un programme comportant des erreurs ne doit pas pouvoir être compilé. Ainsi les erreurs ne risquent pas d'échapper au programmeur et de passer les procédures de tests. En détectant les erreurs à la source, on évite qu'elles se propagent en s'amplifiant.

Ø **Java est un langage compilé :** C'est-à-dire qu'avant d'être exécuté, il doit être traduit dans le langage de la machine sur laquelle il doit fonctionner. Cependant, contrairement à de nombreux compilateurs, java traduit le code source dans le langage de sa JVM. Le code traduit appelé **byte code**, ne peut pas être exécuté directement par le processeur d'une machine.

Ø **Java est doté de standard de bibliothèques de classes :** ces classes sont très riches et elles comprennent la gestion des interfaces graphiques (fenêtres, boites de dialogue, contrôles, menus, graphisme), la programmation multi-threads (multitâches), la gestion des exceptions, les accès aux fichiers… l'utilisation de ces bibliothèques facilite grandement la tâche au programmeur lors de sa construction d'applications complexes.

III-2-2- L'environnement de développement Eclipse IDE :

Eclipse est un projet open source fondé par SUN Microsystems. L'IDE Eclipse est un environnement de développement permettant d'écrire, de compiler, de déboguer et de déployer des programmes. Il est écrit en java, et il y'a un grand nombre de modules pour étendre l'IDE Eclipse.

L'IDE Eclipse est un produit gratuit, sans aucune restriction quant à son usage. L'installation de L'IDE Eclipse nécessite l'installation de la JDK (Java Développement Kit), le kit de développement java compatible avec la version d'IDE. La spécificité d'Eclipse IDE vient du fait que son architecture est totalement développée autour de la notion de **plug-in**. Cela signifie que toutes les fonctionnalités de celui-ci sont développées en tant que plug-in.

Pour concevoir notre application, nous avons utilisé la version Eclipse Galileo. Son interface principale est donnée dans la Figure qui suit :

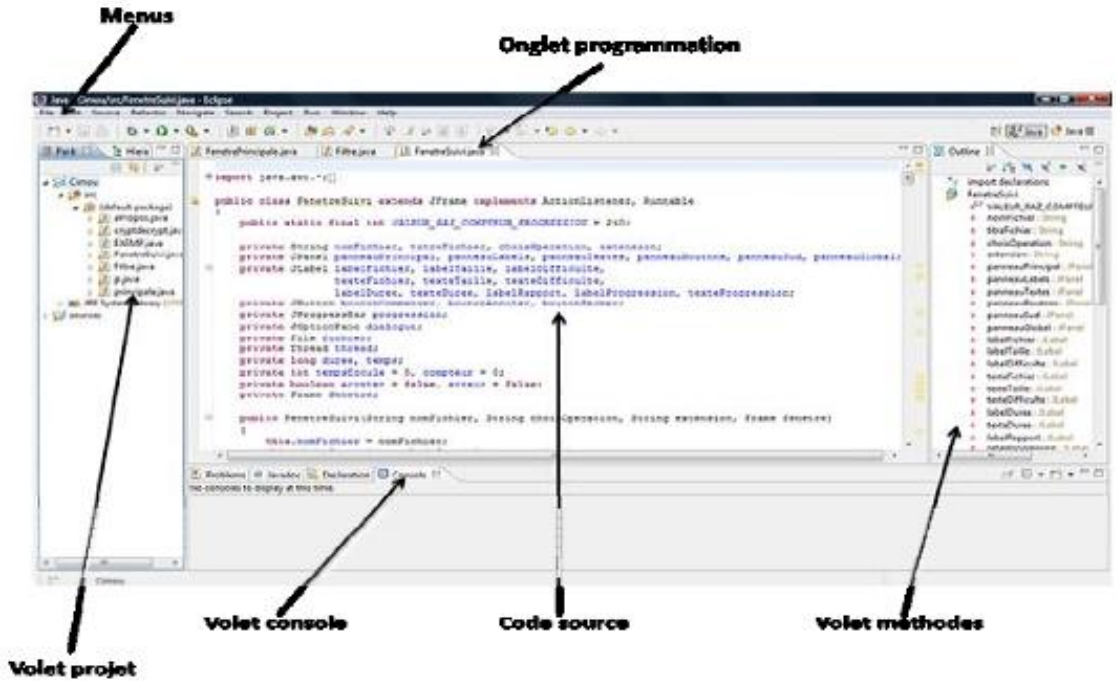


Figure 4-1: Interface JAVA Eclipse.

III-2-3- Parallel Task: le plug-in utilisé [Web 08]

Le plug-in ParaTask a été développé pour Eclipse, vous pouvez la télécharger à partir de l'adresse suivante :

http://www.ece.auckland.ac.nz/~parallelIT/updatesite/

III-2-3-1- Installation du plug-in :

1. Ouvrir Eclipse
2. Cliquer sur help>>>install New Software
3. Cliquer sur Add

4. Copier et collez l'adressee URL du plug-in Para Task dans le champ texte
5. Cliquer sur OK
6. Sélectionner PTjava Text Editor, ensuite cliquez sur Next pour installer le plug-in.

Aprés l'installation de ParaTask, redémarrez Eclipse pour qu'il prenne en considération le nouveau plug-in installé.

III-2-3-2- Création d'un projet PTjava :

Un projet PTjava est tout simplement un projet java en lui associant une spécification PTjava liée au projet. Donc, créer un projet PTjava, est comme créer un projet java simple :

File >> New >> project maintenant au lieu de choisir « java Project » on prend « PTjava Project », et c'est la même chose pour les classes, au lieu de choisir «Class » on prend « PTjava Class ».

IV-Détection de contours d'image :

Tout traitement d'images consiste à améliorer sa qualité, en corrigeant les dégradations subies lors de son acquisition. Une des étapes importante dans le traitement d'images est la détection de contours, car une grande partie de l'information de l'image d'un objet est portée sur les contours de cette dernière, de plus cela permet de sauvegarder le maximum d'informations contenues dans l'image.

L'extraction de contours joue un rôle primordial dans tout système de vision par ordinateur. Beaucoup d'efforts ont été faits pour extraire les contours d'une image et plusieurs techniques ont été proposées à ce jour. Parmi ces dernières le filtre de Rachid DERICHE qui sera détaillé dans ce qui suit.

IV-1- Quelques notions liées à l'imagerie :

Ø Notion de voisinage :

A l'exception des bords d'une image numérique chaque pixel de coordonnées (x, y) à deux voisins horizontaux de coordonnées (x ±1, y), deux voisins verticaux de coordonnées (x, y±1) et quatre voisins diagonaux de cordonnées (x ±1, y±1).

Nous distinguons deux types de voisinages:

- Le 4-voisinage.
- Le 8-voisinage.

Le 4-voisinage d'un pixel de coordonnées (x, y) est constitué de quatre pixels de coordonnées (x, y±1) et (x ±1, y).

Le 8-voisinage d'un pixel de coordonnées (x, y) est constitué de quatre pixels de coordonnées (x, y±1) et (x ±1, y), ainsi que quatre pixels diagonaux de coordonnées (x±1, y±1).

Ø Notion de filtrage :

Le filtrage consiste à appliquer une transformation à toute l'image ou une partie de celle-ci en appliquant un opérateur.

Un filtre est une transformation mathématique (appelée *produit de convolution*) permettant, pour chaque pixel de la zone à laquelle il s'applique, de modifier sa valeur en fonction des valeurs des pixels avoisinants, affectées de coefficients.

Le filtre est représenté par un tableau (matrice), caractérisé par ses dimensions et ses coefficients, dont le centre correspond au pixel concerné. Les coefficients du tableau déterminent les propriétés du filtre. Voici un exemple de filtre 3 x 3 :

1	1	1
1	4	1
1	1	1

Ainsi le produit de la matrice image, généralement très grande car représentant l'image initiale (Tableau de pixels) par le filtre donne une matrice correspondant à l'image traitée.

Ø Image en niveau de gris :

Une image en niveau de gris est composée de 256 niveaux de gris, allant de 0 qui représente le noir absolu jusqu'au le niveau 255 qui représente le blanc parfait.

Ø Bruit : (parasite)

Le bruit caractérise les parasites ou interférences d'un signal, c'est-à-dire les parties du signal déformées localement. Ainsi le bruit d'une image désigne les pixels de l'image dont l'intensité est très différente de celles des pixels voisins.

IV-2- Présentation du filtre de R- Deriche :

Deriche s'est focalisé sur l'extraction des contours d'images dans ces recherches. Il a abouti à un filtre à réponse impulsionnelle infinie pour le calcul de la dérivée. [INRIA Deriche]

$$d(x) = c\, x\, e^{-\propto|x|} \text{ avec } c = -\frac{(1-e^{-\propto|x|})}{e^{-\propto}}$$

Les opérateurs de Deriche sont utilisés dans deux grandes méthodes de détection de contours : l'une basée sur les maxima locaux du gradient, l'autre sur les passages par zéro du laplacien. Les méthodes récentes combinent le gradient et le laplacien. Les opérateurs implémentés sont ceux proposés par Rachid Deriche. [Deriche EALG]

Dans notre cas, on s'est focalisé sur la méthode des maxima locaux du gradient.

IV-2-1- Les différentes étapes du filtre de Deriche : [Deriche EALG]

a- Le lissage :

Le lissage est un pré filtrage réalisé par un filtre passe bas[1], c'est une opération primordiale dans la détection des contours d'images numériques, et est une solution très efficace pour palier aux problèmes causés par les bruits lors de la détection des contours, car ces dernier peuvent être pris pour des contours lors du traitement.

Afin de réaliser l'opération de lissage on utilise le filtre suivant :

$$s(x) = b(\propto |x| + 1)e^{-\propto|x|}$$

Ou $$b = (1 - e^{-\propto|x|})^2/(1+\propto e^{-\propto} - e^{-2\propto})$$

b- Gradient de Deriche :

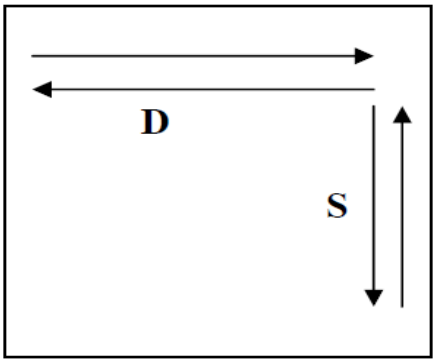
La dérivée directionnelle selon x est le résultat d'un lissage suivant la direction y, suivi d'une dérivation suivant x. Pour la dérivée suivant y, on permute les directions.

Gradient en x : $\boldsymbol{I_x(x,y) = \big(Im * s(y)\big) * d(x)}$

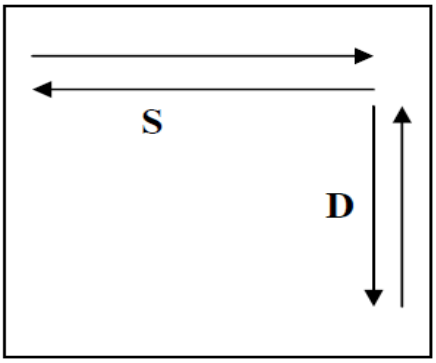
Gradient en y : $\boldsymbol{I_y\,(x,y) = \big(Im * s(x)\big) * d(y)}$

Où **d(x) :** est le filtre de dérivation
s(y) : le filtre de lissage

[1] Consistant à atténuer les composantes de l'image ayant une fréquence haute (pixels foncés). Ce type de filtrage est généralement utilisé pour atténuer le bruit de l'image, c'est la raison pour laquelle on parle habituellement de lissage. Le résultat de ce filtre est une image plus floue.



Calcul du gradient selon x



Calcul du gradient selon y

Figure 4-2: Calcul du gradient.

c- Extraction des maxima locaux :

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

Cette étape vise bien la diminution de l'épaisseur des contours générer par l'étape précédente

e- Le seuillage :

L'étape de seuillage vient après l'extraction des maxima locaux, elle consiste à coder l'image résultante de l'opération précédente sur deux nuances de gris (le noir et le blanc généralement), par l'établissement d'un seuil, puis de coder les points qui en un niveau de gris supérieur au seuil par un des deux nuances de gris précédentes, et les autres par l'autre nuance.

Le seuil est établi ont appliquant une méthode statistique qui consiste à calculer la moyenne des niveaux de gris des points maxima locaux sans considérer le fond de l'image.
Le seuil « S » est calculé par cette formule :

$$S = \frac{\sum_{i=0}^{L-1} \sum_{j=0}^{C-1} N_{ML}(i,j)}{M}$$

Où :

NG : le niveau de gris du point maxima locaux ;
M : le nombre de point non nuls dans l'image maxima locaux ;
c : le nombre de colonne de l'image ;
l : le nombre de ligne de l'image.

IV-2-2- Implémentation du filtre de R- DERIHE : [TP MPI]

IV-2-2-1-Opérateurs bidimensionnels de dérivation et de lissage de Deriche:

d (x) = $cx\ e^{-\propto|x|}$ avec [1]

C= $(1 - e^{-\propto|x|})^2/e^{-\propto}$

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

f(x)= b($\propto |x| + 1$) $e^{-\propto|x|}$ où b= $\left(1 - e^{-\propto|x|}\right)^2/(1 + 2 \propto e^{-\propto} - e^{-2\propto})$

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

a- Le lissage :

La transformée en Z du filtre correspondant à l'expression [1] est :

f[z]= $f_-[Z^{-1}]$ + f_+ [z] avec

$$f_-(Z^{-1}) = (1+e^{-\propto}(\propto -1)Z^{-1})/\ (1-2e^{-\propto}Z^{-1} + e^{-2\propto}Z^{-2})$$

$$f_+(Z) = be^{-\propto}(\propto +1)Z + e^{-2\propto}Z^2)/\ (1-2e^{-\propto}Z + e^{-2\propto}Z^2)$$

Les équations aux différences auront alors pour expressions:

B_1 [i]=b A[i]+b $e^{-\propto}(\propto -1)$ A[i-1]+2$e^{-\propto}B_1$ [i-1] - 2$e^{-2\propto}B_1$[i-2] pour 1..M

B_2 [i]=b $e^{-\propto}$ (□ 1) A[i+1]-b $e^{-2\propto}$ A[i+2]+2$e^{-\propto}$ B_2[i+1] -$e^{-2\propto}$ B_2 [i+2] pour M..1

Le signal filtré résultant est : **B[i]=B1[i]+B2[i]**.

Dans ce cas on a 8 multiplications et 7 additions.

b- Opérateur gradient bidimensionnel :

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

$$f(n) = [(c_1 \sin(\omega|n|) + c_2(\cos(\omega|n|)]\, e^{-\propto|n|}$$

Avec

F(n)= $f_-(n)$ + $f_+(n)$

Sa transformer en Z est :

f(Z) = $f_{-}(Z) + f_{+}(Z)$

Donc

$$f_{-}(Z) = (a_0 + a_1 Z^{-1})/(1 + b_1 Z^{-1} + b_2 Z^{-2})$$

$$f_{+}(Z) = (a_2 + a_3 Z^{2})/(1 + b_1 Z^{1} + b_2 Z^{2})$$

Avec

$$a_0 = c_2$$
$$a_1 = c_1[\sin\omega - c_2 \cos\omega]e^{-\propto}$$
$$a_2 = a_1 - c_2$$
$$b_1 = -2e^{-\propto}\cos\omega$$
$$b_2 = e^{-2\propto}$$

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

$$\frac{R_x(m,n)}{X(m,n)} = \frac{\left[-ce^{-\propto|m|}\,\S\ddot{o}\ddot{u}\,(\omega|m|)\right][k \propto \S\ddot{o}\ddot{u}\,(\omega|n|)\omega\hat{\imath} \dagger \S(\omega\,|n|)e^{-\propto|n|}]}{\propto^2 + \omega^2}$$

$$\frac{R_Y(m,n)}{Y(m,n)} = \frac{\left[-ce^{-\propto|n|}\,\S\ddot{o}\ddot{u}\,(\omega|n|)\right][k \propto \S\ddot{o}\ddot{u}\,(\omega|m|)\omega\hat{\imath} \dagger \S(\omega\,|m|)e^{-\propto|m|}]}{\propto^2 + \omega^2}$$

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

$$y_{+}(i,j) = A(i,j-1) - b_1 y_{+}(i,j-1) - b_2 y_{+}(i,j-2)$$

$$y_{-}(i,j) = A(i,j+1) - b_1 y_{-}(i,j+1) - b_2 y_{-}(i,j+2)$$

Avec

$$\begin{cases} j = 1..n \;\; colonnes \\ i = 1..n \quad lignes \end{cases}$$

La matrice de lissage est donnée par :

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

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

$$R_x(i,j) = a_2 y(i+1,j) + a_3 y(i+2,j)$$

Où j= 1…n colonnes et i = 1…..m lignes

D'ou la matrice du gradient directionnel :

$$R_x(i,j) = R_{x+}(i,j) + R_{x-}(i,j)$$

Les coefficients utilisés ci dessus sont les mêmes que ceux définis précédemment. On applique la même procédure pour déterminer $R_y(i,j)$

Les filtres présentés permettent de calculer le gradient mais ne donnent pas les points de contour. Des post traitements sont nécessaires pour l'image gradient tel que :

IV-2-2-2-Extraction des maxima locaux de la norme du gradient :

L'algorithme d'extraction des maxima locaux consiste à comparer la norme du gradient $N_G(i,j)$ du point M(i,j) à celle de ses deux voisins N_G1 du point M1 et N_G2 du point M2 .

La norme du gradient est donnée par l'expression :

$$N_G(i,j) = \sqrt{\#©(ï,ð) + \#™(ï,ð)}$$

Le point M est un maxima local si :

$$N_G > N_G1 \text{ et } N_G > N_G2$$

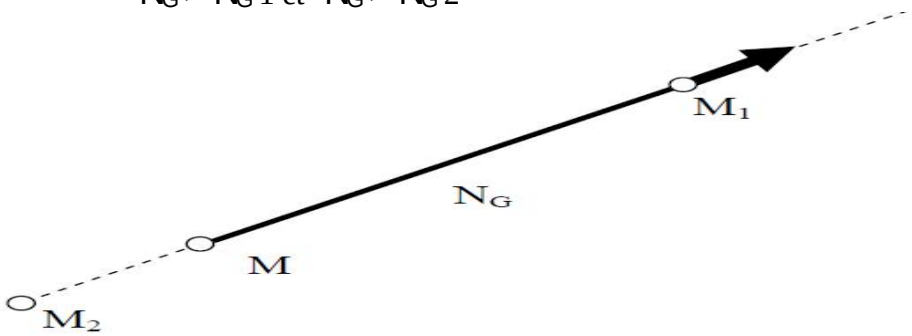


Figure 4-3: Extraction du maxima local du point M

IV-2-2-3-Seuillage :

Après extraction des maxima locaux, une autre opération s'applique à l'image : c'est le seuillage. Son rôle est de produire une image dont les pixels ne sont codés que sur deux nuances de gris: le noir prend la valeur 0 et le blanc prend la valeur 255.

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

$$S = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} N_G((i,j))$$

N_G : le niveau de gris du point maxima local.
M : le nombre de points non nuls dans l'image maxima locaux.
$N_G(i,j) = 255$ pour $N_G(i,j) >= S$ ou $N_G(i,j) = 0$ pour $N_G(i,j) < S$
n: le nombre de colonne de l'image ;
m: le nombre de ligne de l'image.

V- Présentation de l'application de détection de contours:

Il s'agit d'une implémentation de l'algorithme de R-DERICHE pour la détection des contours d'image, l'application se présente comme une fenêtre GUI, permettant à l'utilisateur de sélectionner une image à partir de son disque dure, ensuite appliquer le filtre de détection de contours, le traitement peut se faire d'une manière séquentielle, parallèle avec une décomposition de données par blocs ou cyclique .

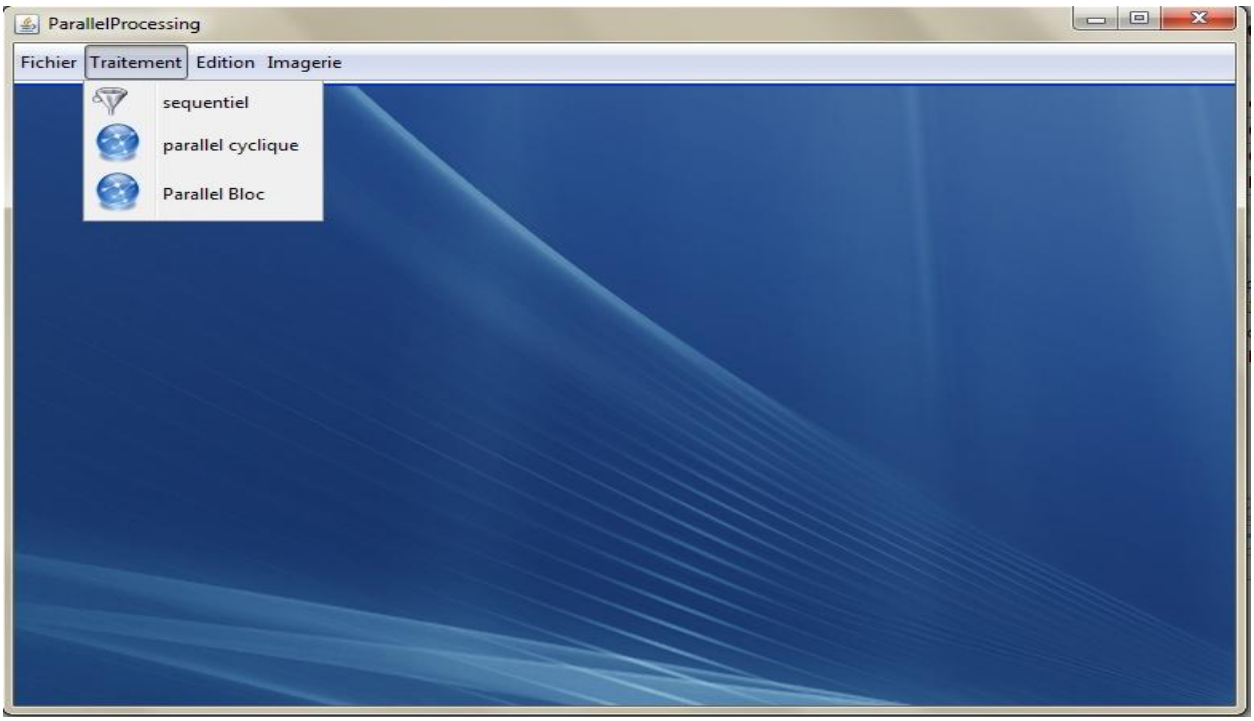


Figure 4-4 : Fenêtre d'accueil de l'application.

V-1- L'exécution séquentielle :

L'exécution séquentielle sur une machine à deux processeurs, a donné les résultats suivants :

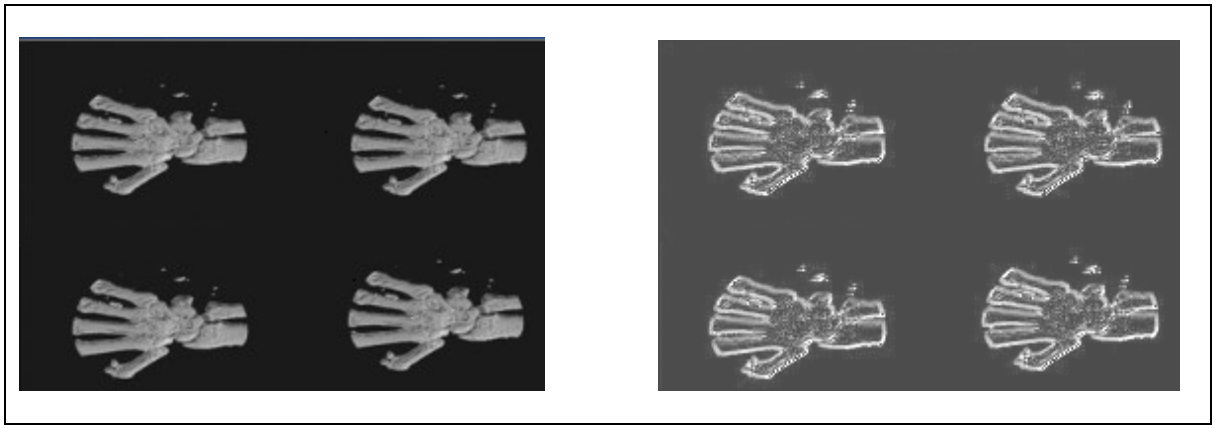


Figure 4-5 : Détection des contours de l'image1.

Le temps de traitement en séquentiel, nécéssaire pour l'image 1 est de : 0.127 secondes.

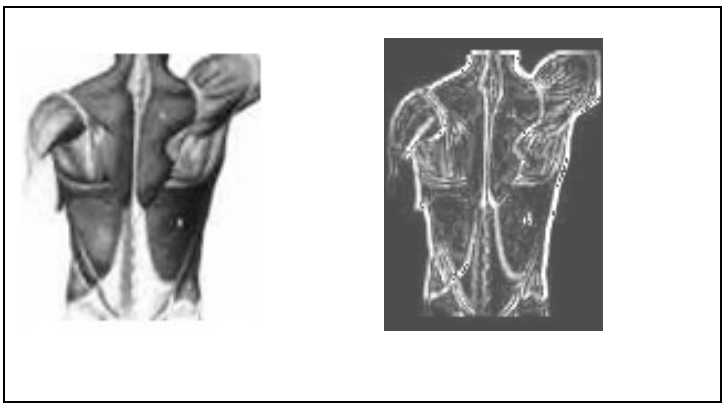


Figure 4-6 : Détection des contours de l'image 2.

Le temps de traitement en séquentiel, nécéssaire pour l'image 2 est de : 0,021 secondes.

V-2- Sollution parallèle :

Dans notre cas nous allons utiliser la décomposition du domaine , c'est-à-dire l'image sera partagée en plusieurs portions, chaqu'une d'elle sera affectée à une tâche qui calcule les contours de la portion de l'image, à la fin du traitement, on récupère les résultats des différentes portions et on affiche l'image finale. Pour cela nous allons utiliser une répartition de données par bloc et répartition cyclique de données.

V-2-1- Répartition de données par bloc :

L'image sera diviser en un nombre égal au nombre de tâches parallèles qu'on estime créer. Chaque tâche aura deux entiers en paramètres, début et fin, qui représentent les pixels à traiter par cette dernière.

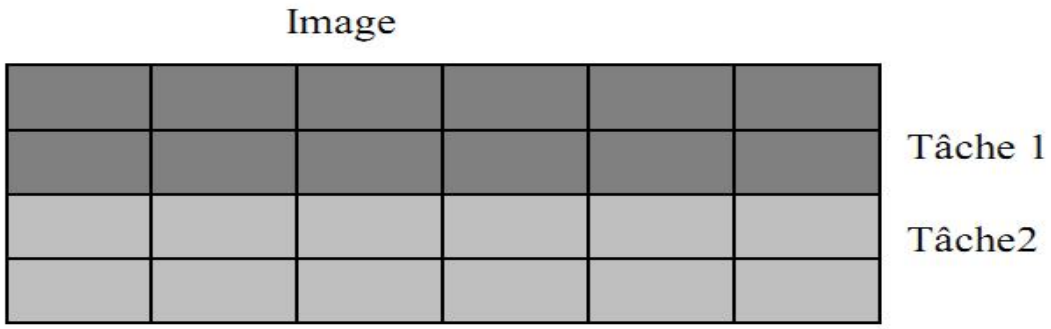


Figure 4-7 : répartition de données par blocs.

On aura qu'une seule définition de tâche, car dans ce cas on applique le même traitement sur des données différentes. La définition de la tâche est la suivante :

```
TASK public static void _pt_taskgris (BufferedImage img,BufferedImage imgr, int debut, int fin)
{
    long start = System.currentTimeMillis(); // Initialiser le temps d'exécution
    filter (img, imgr, debut , fin);
    long time = System.currentTimeMillis() - start; // Calculer le temps d'éxecution
    long id = Thread.currentThread().getId(); // Récupérer le thread en court
    System.out.println(time / 1000.0 + " seconds, pour thread["+id+"]");

    }
```

Dans la fonction filter, **img** représente l'image qu'on veut traiter et **imgr** l'image résultat.

Après avoir défini la tâche à exécuter en parallèle, il ne reste plus qu'à l'invoquer. Le nombre de tâches à invoquer dépend en faite du matériel utilisé (nombre de processeurs). Heureusement avec ParaTask, même si on invoque un nombre de tâches supérieur au nombre de processeurs, l'application va s'exécuter normalement et dans ce cas le processeur fera du multitâche.

Deux tâches seront invoquées dans notre cas:

```
TaskIDGroup g = new TaskIDGroup (2);
    TaskID id1 = taskgris (img, imgr, 0, h / 2 + 3);
      g.add(id1);
```

```
TaskID id2 = taskgris (img, imgr, h / 2, h);
    g.add (id2);
```

L'exécution de ce programme à donné les résultats de détection de contours suivant : On donnera d'abord les résultats retournés par chaque processeur séparément ensuite le résultat total.

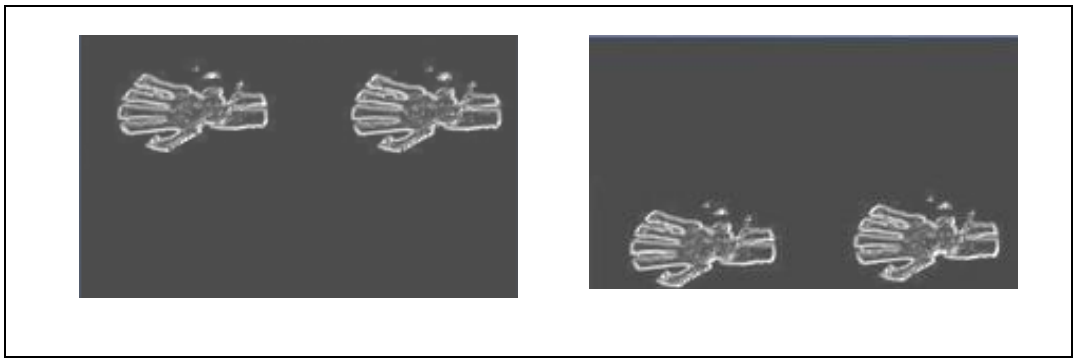


Figure 4-8 : Décomposition par bloc de l'image 1.

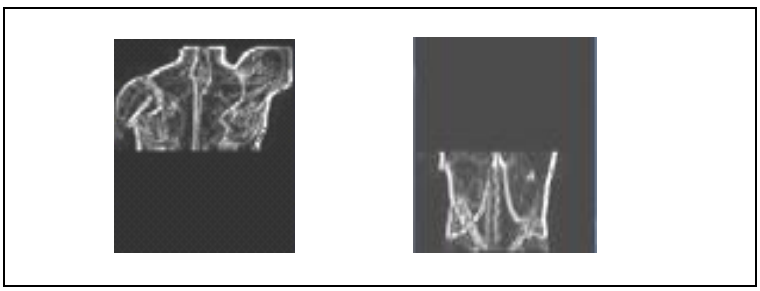


Figure 4-9 : Décomposition par bloc de l'image 2.

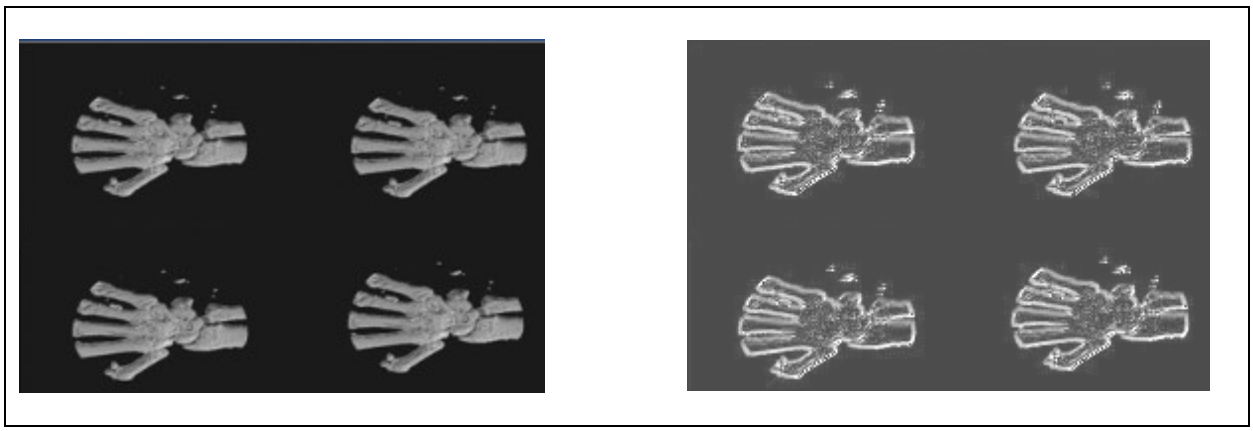


Figure 4-10: Décomposition par bloc de l'image 1 (entière).

Temps d'exécution pour chaque thread est : 0.103 secondes pour le thread [25]

0.104 secondes pour le thread [26]

Le temps d'exécution final est le maximum entre (0.103, 0.104) plus le temps de reconstitution de l'image qui est égale à 0.002 secondes, c'est à dire 0.104+0.002=0.106 secondes.

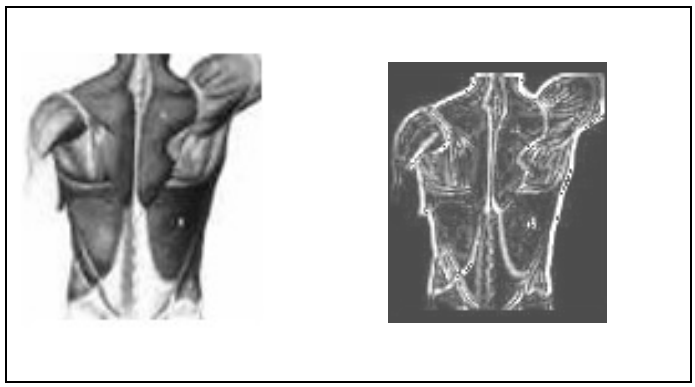


Figure 4-11: Décomposition par bloc de l'image 2 (entière).

Temps d'exécution pour chaque thread est : 0.011 secondes pour le thread [25]

0.009 secondes pour le thread [26]

Le temps d'exécution final est le maximum (0.011, 0.009) plus le temps de reconstitution de l'image qui est égale à 0.001 secondes, c'est à dire : 0,011+0.001=0.012 secondes.

V-2-2- Répartition cyclique de données :

De la même manière que la répartition par bloc, l'image sera décomposée en plusieurs portions, mais cycliquement comme le montre le schéma suivant :

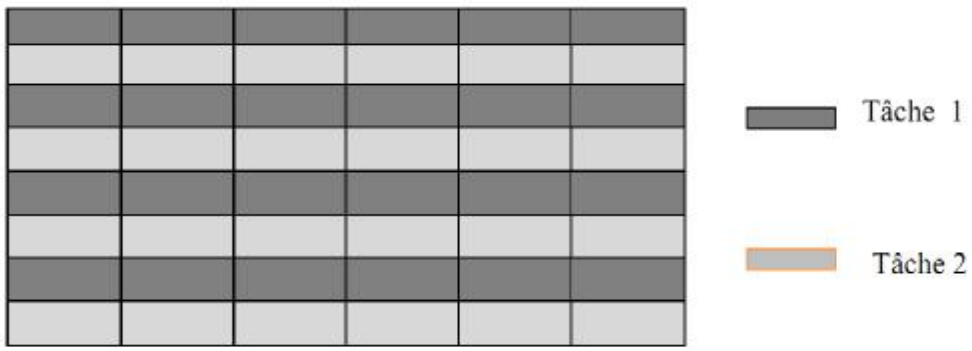


Figure 4-12 : Répartition cyclique des données d'une image.

La définition de la tâche parallèle reste la même sauf en ce qui concerne la façon de définir les entiers « débuts » et « fin » de la portion de pixels traitée par une tâche. Ces entiers vont être calculés **n** fois pour chaque tâche avec **n** égal au nombre de cycle qu'on estime créer pour chaque tâche. Dans notre cas **n**= 4.

```
TASK public static void _pt_CYCtaskgris (BufferedImage img,BufferedImage
imgr, int h,int x)
{
int debut=0; int fin=0;
for (int i=0; i<4; i++)
{
debut=(2*i+x)*h/8;
if (i==3 & x==1)
fin=(2*i+x+1)*h/8;
else
fin=(2*i+x+1)*h/8+3;
long start = System.currentTimeMillis();
Contour.filter(img,imgr,debut,fin);
long time = System.currentTimeMillis() - start;
long id = Thread.currentThread().getId();
System.out.println(time / 1000.0 + " seconds, pour thread["+id+"]"+"cycle"+i);
 } }
```

Après avoir défini la tâche parallèle à exécuter, il ne reste plus qu'à l'invoquer:

```
TaskIDGroup g=new TaskIDGroup(2);
TaskID id1 =CYCtaskgris (img, imgr, h, 0);
g.add(id1);
TaskID id2 = CYCtaskgris (img, imgr, h, 1);
g.add(id2) ;
```

L'exécution de ce programme a donnée les résultats de détection de contours suivant :
On donnera d'abord les résultats retournés par chaque processeur séparément ensuite le résultat total.

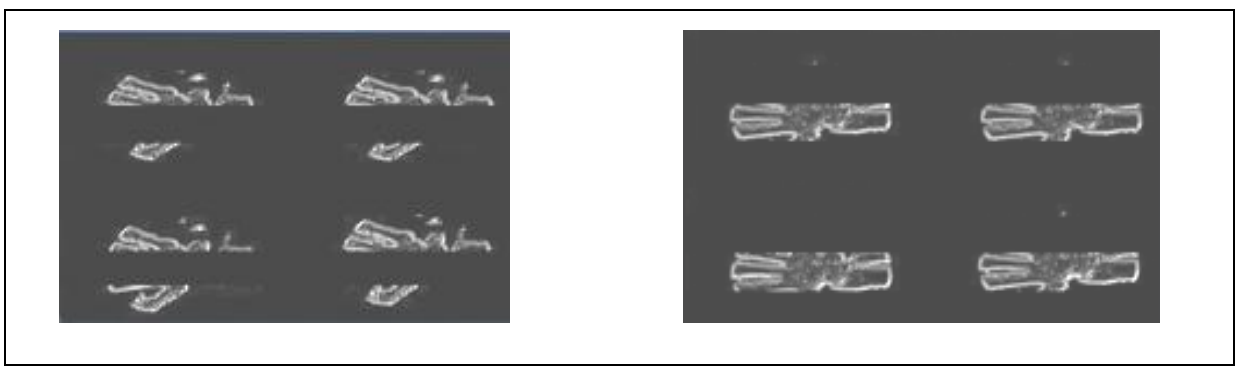


Figure 4-13 : Répartition cyclique de données de l'image 1

L'exécution sur l'image entière :

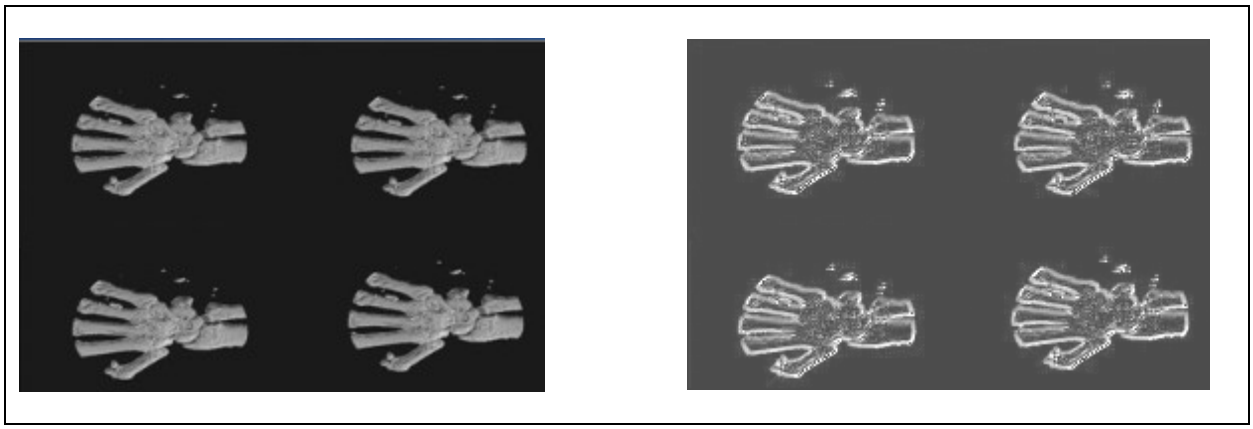


Figure 4-14 : Répartition cyclique de données de l'image 1 (entière).

L'exécution parallèle sur l'image 1 entière nous donne les résultats suivants :

Temps (seconde)	Thread	Portion
0 .011	25	0
0.012	26	0
0.011	25	1
0.013	26	1
0.072	25	2
0.075	26	2
0.011	25	3
0.009	26	3

Le temps d'exécution du thread [25] est : 0.011+0.011+0.072+0.011= 0.105 secondes.

Le temps d'exécution du thread [26] est : 0 .012+0.013+0.075+0.009= 0.109 secondes.

Le temps d'exécution total de l'image 1 est le max (0.105, 0.109) plus le temps de reconstitution de l'image qui est égale à 0.002 secondes: 0.109+0.002=0.111 secondes.

On procédera de la même manière que l'image 1 pour la répartition cyclique de données sur l'image 2, donc on présentera l'exécution de chaque processeur, après l'exécution de l'image entière :

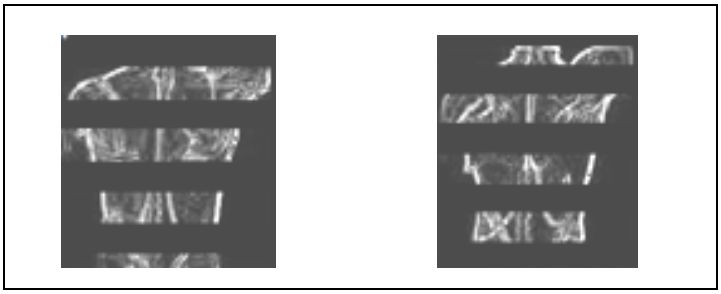


Figure 4-15 : Répartition cyclique de données de l'image 2.

L'exécution sur l'image entière :

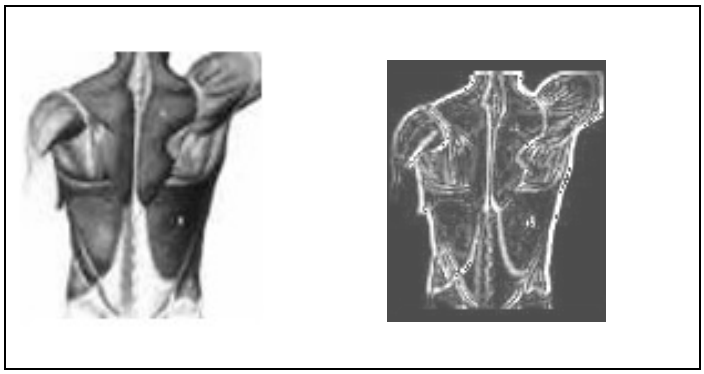


Figure 4-16 : Répartition cyclique de données de l'image 2 (entière).

L'exécution parallèle sur l'image 2 entière nous donne les résultats suivants :

Temps (second)	Thread	Portion
0.004	34	0
0.004	35	0
0.004	34	1
0.004	35	1
0.003	34	2
0.003	35	2
0.003	34	3
0.004	35	3

Le temps d'exécution du thread [29] est : 0.004+0.004+0.003+0.003= 0.014 secondes.

Le temps d'exécution du thread [30] est : 0.004+0.004+0.003+0.004 = 0.015 secondes.

Le temps d'exécution final de l'image 2 est le maximum entre (0.014,0.015) plus le temps de reconstitution de l'image qui est égale à 0.001 secondes : 0.015+0.001=0.016 secondes.

VI- Calcul de l'accélération et de l'efficacité :

Pour récapituler les différents traitements que nous avons réalisé auparavant, nous allons représenter les différents résultats dans un tableau récapitulatif en calculant l'accélération et de l'efficacité. Avec :

Accélération= temps d'exécution séquentiel / temps d'exécution parallèle

Efficacité= Accélération/ nombre de processeurs.

	Image 1	Image 2
Temps d'exécution séquentiel	0.127	0.021
Temps parallèle (bloc)	0.106	0.012
Accélération (bloc)	1.199	1.750
Efficacité (bloc)	0.600	0.954
Temps parallèle (cyclique)	0.111	0.016
Accélération (cyclique)	1.144	1.312
Efficacité (cyclique)	0.572	0.656

Figure 4-17 : Tableau récapitulatif des temps d'exécutions de chaque solution.

VII-Conclusion

Nous avons vu dans ce chapitre, la manière dont on déclare et on invoque des tâches parallèles, et cela à travers des exemples que nous avons présentés, malgré qu'ils ne soient pas assez compliqués et qui nécessitent un traitement parallèle.

Nous avons aussi déduit que, ParaTask est bien utile dans la construction des applications desktop parallèle, qui nécessitent de la synchronisation et de la communication entre les tâches ainsi que des communications avec l'extérieur.

Pour conclure on dit que, vue sa facilité d'utilisation, ParaTask constitue une bonne solution pratique pour construire des applications GUI parallèles.

Au cours de ce mémoire, nous avons vue comment l'augmentation du nombre de processeurs des micro-ordinateurs a influencé sur la conception des applications desktop parallèles.

Le choix de la bibliothèque parallèle « ParaTask » choisie parmi plusieurs outils y inclus les trois qu'on à étudier ; nous a été conseillé, car tous ces outils facilitent d'une façon remarquable la construction de ce type d'applications d'un coté, et d'un autre accélèrent la vitesse d'exécution sans toucher à la qualité du logiciel.

Ce travail nous a permit de paralléliser l'extraction de contour d'images, en utilisant l'outil choisi. Le traitement parallèle obtenu sur deux images de tailles différentes, ont permis d'atteindre une accélération qui vari entre 1.144 et 1.750 ; avec une efficacité qui varie entre 0.572 et 0.954 par rapport au traitement séquentiel.

Nous pouvons dire que ces résultats obtenus sont bons et tendent vers l'accélération théorique que l'on peut atteindre sur des systèmes à deux processeurs.

Pour conclure, la réalisation d'une telle application nous a donné l'occasion d'acquérir d'intéressantes connaissances concernant le parallélisme, nous avons aussi eu l'occasion d'apprendre à paralléliser un code séquentiel et vu quel intérêt apporte le traitement parallèle par rapport à un traitement séquentiel, celui de minimiser et d'accélérer le temps d'exécution de ce dernier.

Bibliographie

[Belkacem 2011] Parallélisme dans les applications desktop , khaled belkacem, 2011.

[BL] Introduction to Parallel Computing Blaise Barney, Lawrence Livermore

[CP 2004] Conception d'un système à haute performance - Le calcul parallèle, 2004

[Deghilage,1992] Architecture et programmation parallèle : approche pratique en environement scientifique sur multiprocesseurs. Paris : Addisson Wesly Blaise Barney, Introduction to Parallel Computing.[en ligne].

[Deriche EALG] Exécution temps réel des détecteurs de contours de Deriche par des processeurs RISC ; Thomas EA, Lionel LACASSAGNE, Patrick GARDA.

[GS, parallel] Nasser Giancaman, Oliver Sinnen , university of Auckland, New Zealand, Task Parallelism for Object Oriented Program.

[INRIA Deriche] Techniques d'extraction de contours d'image, Rachid DERICHE, INRIA Sophia-Antipolis

[JAVA EYROLLES] programmer en JAVA, Claude DELANNOY, EYROLLES,5 édition java 5 et 6

[Marchand MPI] Les architectures parallèles, Les machines parallèles de l'observatoire MPI, Aurélia Marchand

[NG,OS] Nasser Giacaman et Oliver Sinnen**,** Object-Oriented Parallelization of Java Desktop Programs. PARALLELISM ON THE DESKTOP, [en ligne] http://www.ece.auckland.ac.nz/~parallel/ParallelIT/index.html.

[PSK 2008] Victor Pankratius, Wolframe Schulte et Kurt Keutze, Parallelism on the Desktop, IEEE 2008

[Vollaire1997] Modélisation de phénomènes électromagnétiques hyperfréquences sur calculateurs parallèles, Christian VOLAIRE, 1997**.**

[WK, MV] Wooyoung Kim et Michael Voss, Intel Multicore Desktop Programming with Intel Threading Building Blocks. Parallelism on the desktop.

[TP MPI] Traitement Parallèle sous MPI-2, pour l'Accélération de l'Algorithme d'Extraction de Contours d'Images. Taguemount A, Djema L, Boumghar F.O.Mars 2005,Tunisie.

Webographie

[Web 00] http://blog.invokk.net/2011/08/parallelisme-memoire-distribuee-et-partagee

[Web 01] www.wikipedia.org

[Web02] Blaise Barney, Introduction to Parallel Computing.[en ligne]. https://computing.llnl.gov/tutorials/parallel_comp/

[Web 04] http://www.cetmef.devellopement-durable.gouv.fr/IMG/pdf/calcul-para.pdf

[Web 05] http://www-id.imag.fr/~traored/**WWW**-documents/these_Daouda.pdf

[Web 06] www.site du zero.com.

[web07] http://software.intel.com/en-us/blogs/2007/09/12/threading-building-blocks-atomic-operations-introduction/

[Web 08] Oliver Sinnen, Nasser Giacaman, Christopher Chong et Lama Akeila. ParallelIT.[enligne].http://www.ece.auckland.ac.nz/~parallel/ParallelIT/index.html.