REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI DE TIZI OUZOU

FACULTE DE GENIE ELECTRIQUE ET DE L'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

Mémoire de fin d'études

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

Spécialité : conduite de projet informatique

Thème :

Parallélisme dans les applications desktop

LOUNES DJEMA BELKACEM KHALED

Promotion : 2010-2011

Remerciements

Nous tenons à remercier vivement notre promoteur Mr Djema Lounes pour nous avoir proposé ce sujet, pour la qualité de son encadrement, et son suivi durant toute la durée du projet.

Nous remercions aussi les membres de jury pour l'honneur qu'ils nous font en acceptant de juger ce mémoire.

Enfin, nous remercions toutes les personnes ayant contribué de prés ou deloin au bon accomplissement de notre travail.

Dédicaces

Je dédie ce modeste travail

A ma mère et a mon père qui me sont les plus chères au monde et auxquels je ne saurais jamais exprimer ma gratitude et ma connaissance en quelques lignes.
A mes frères.
A mes sœurs.
A tous mes amis, particulièrement Ahmed, Moussa et Nassim avec lesquels j'ai passé mes plus beaux jours.

BELKACEM Khaled

SOMMAIRE

Liste des figures

Introduction générale

La construction des micro-ordinateurs à connue un grand défie ces dernières années, améliorer la vitesse d'exécution d'un processeur revient plus chère que d'en construire un autre, et si on se réfère à la loi de moor la vitesse d'exécution se multiplie par deux chaque 18 mois, par conséquent le nombre de processeurs va engrener de plus en plus pour les futures ordinateurs.

Cette augmentation du nombre de processeurs a amener les chercheurs à repenser l'ingénierie du logiciel qui à était auparavant séquentiel et introduire une nouvelle ingénierie du logiciel parallèle qui exploitera d'une façon efficace les ordinateurs parallèle.

Ce mémoire rentre dans les cadre du parallélisme dans les applications desktop, le chapitre I donne des généralités sur les ordinateurs et la programmation parallèle, le chapitre II traite la démarche à suivre pour la conception un programme parallèle, le chapitre III est consacré pour les outils (bibliothèques) pour le développement d'applications desktop parallèle, et enfin le chapitre IV qui est consacré pour mieux expliquer la bibliothèque parallèle de java (ParaTask) et donner quelques exemples de programmation.

Chapitre I

Généralités sur le parallélisme et la programmation parallèle

Introduction :

Autrefois, le logiciel s'exécute d'une façon séquentielle, dans un ordinateur à une seule unité centrale de calcul (CPU), une instruction à la fois.

Mais avec le calcul parallèle, 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.

Les ressources de calcul peuvent être un ordinateur à plusieurs processeurs, un ensemble d'ordinateurs connectés par un réseau ou une combinaison des deux.

Le but de ce chapitre est d'introduire la notion du parallélisme et la programmation parallèle et d'expliquer la nécessité de l'exploiter pour atteindre de hautes performances.

I. Concepts et terminologies :

I.1. l'architecture de Von Neumann : [2]

Cette architecture est nommée d'après le mathématicien John von Neumann qui a soumis la première description d'un ordinateur dont le programme est stocké dans sa mémoire. Le terme « architecture de Von Neumann » est néanmoins considéré comme injuste vis-à-vis des collaborateurs de John Von Neumann, notamment John William Mauchly et John Eckert qui ont utilisé ce concept pendant leurs travaux sur l'ENIAC et il est donc maintenant plutôt proscrit.

1. L'unité arithmétique et logique ou unité de traitement : son rôle est d'effectuer les opérations de base ;
2. L'unité de contrôle, chargée du séquençage des opérations ;
3. La mémoire qui contient à la fois les données et le programme. La mémoire se divise entre mémoire volatile RAM (programmes et données en cours de fonctionnement) et mémoire permanente ROM (programmes et données de base de la machine).
4. Les dispositifs d'entrée-sortie, qui permettent de communiquer avec le monde extérieur.

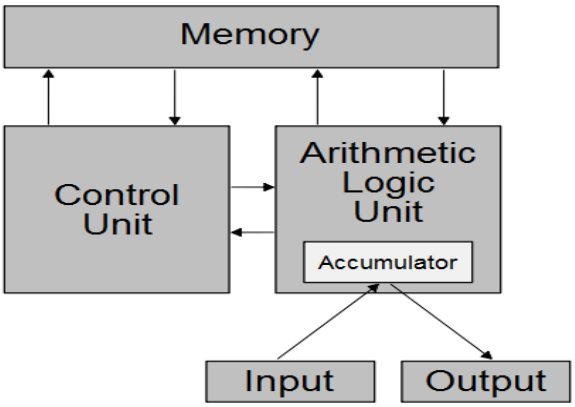


Figure 1.1 machine de Von Neumann

I.2. Classification des ordinateurs parallèles : [2]

Ils existent différentes façons de classification des ordinateurs parallèles, la plus utilisée est la classification de Flynn depuis 1966.

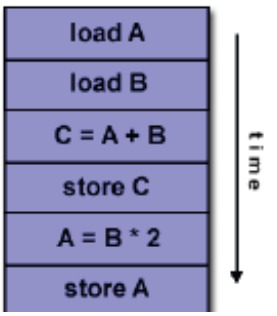
Cette classification classifie les architectures d'ordinateurs multiprocesseur selon deux critères les instructions et les données.

SISD une instruction, une donnée	SIMD une instruction, données multiples
MISD instructions multiples, une seule donnée	MIMD instructions multiples, données multiples

Figure 1.2 classification des ordinateurs parallèles

- ***Single Instruction, Single Data (SISD):***

Machine de von Neumann non parallèle. La plupart des ordinateurs séquentiels relèvent de cette catégorie.



Un ordinateur de type SISD peut éventuellement avoir plusieurs unités fonctionnelles de traitement, mais elles sont dirigées par une seule unité de contrôle.

Figure 1.3 Single Instruction, Single Data (SISD)

- ***Single Instruction on Multiple Data(SIMD)*** :

Désigne un mode de fonctionnement des ordinateurs dotés de plusieurs unités de calcul fonctionnant en parallèle. Dans ce mode, la même instruction est appliquée simultanément à plusieurs données pour produire plusieurs résultats. Le modèle SIMD convient particulièrement aux traitements dont la structure est très régulière, comme c'est le cas pour le calcul matriciel.

Les instructions SIMD ont été ajoutées aux processeurs modernes pour pouvoir améliorer la vitesse de traitement sur les calculs impliquant des nombres en virgule flottantes

On traite les mêmes instructions en parallèle, puis on passe au calcul suivant lorsque les deux ont fini.

Dans le cas du SIMD parallèle, on traite une instruction en exécution asynchrone.

En général, pour écrire un programme on commence par développer un code générique qui fonctionnera partout. Quand l'algorithme est correct et que le code fonctionne bien, on écrit une version spécialisée pour une extension d'un processeur donné.

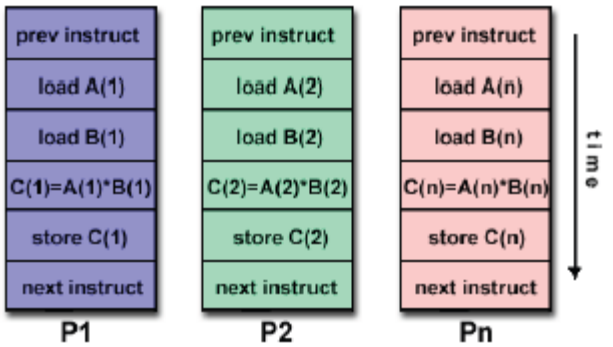


Figure 1.4 Single Instruction on Multiple Data(SIMD)

- ***Multiple Instruction,Single Data (MISD) :***

Il s'agit d'un ordinateur dans lequel une même donnée est traitée par plusieurs processeurs en parallèle. Il existe peu d'implémentations en pratique. Cette catégorie peut être utilisée dans le filtrage numérique et la vérification de redondance dans les systèmes critiques.

On pourrait considérer les processeurs à architecture pipeline comme MISD, en effet l'unité de traitement pipeline traite la même donnée sur différents étages correspondant respectivement à différents partie d'instructions.

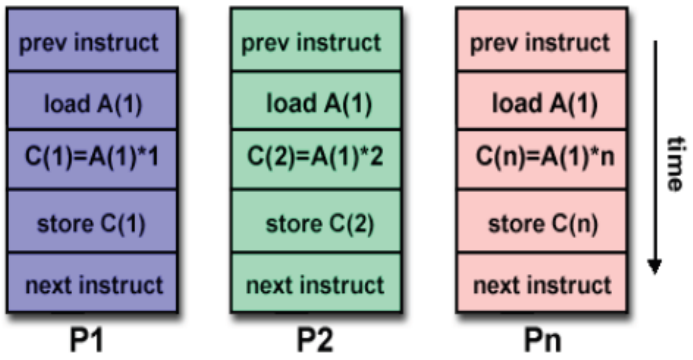


Figure 1.5 Multiple Instruction,Single Data (MISD)

- ***Multiple Instruction,Multiple Data (MIMD) :***

Cette catégorie utilise plusieurs flux de contrôle et plusieurs flux de données, l'idée de base consiste à interconnecter un certain nombre d'ordinateurs complets, l'utilisateur utilise autant d'unités qu'il désire de puissance. Cette architecture est plus générale ; puisqu'elle permet le traitement parallèle d'une application ou le multitraitement de plusieurs applications simultanément. La caractéristique principale de cette architecture repose sur la façon dont l'information est partagée : mémoire centrale partagé (multiprocesseur) ou communication par passage de message (multiordinateur).

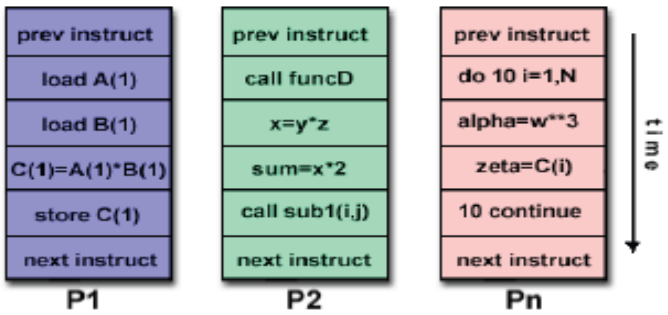


Figure 1.6 *Multiple Instruction, Multiple Data (MIMD)*

II. les architectures mémoire d'un ordinateur parallèle [3]

II.1. la mémoire partagée :

II.1.1 caractéristiques générales:

La mémoire partagée désigne un large bloc de mémoire vive qui est accédé par différentes unités de calcul au sein d'un ordinateur parallèle. Le problème sous-jacent est la vitesse d'accès aux données, et les différents niveaux de cache que possède l'architecture. Quand une donnée est mise à jour et qu'elle est utilisée par différentes unités de calcul, cette modification doit être répercutée dans tous les « caches » ; dans le cas contraire, les unités de calcul travailleraient à partir de données incohérentes. Ce phénomène est connu sous le nom de cohérence de cache ou plus généralement de cohérence de mémoire.

Ils existent différents types de mémoire partagée pour les ordinateurs parallèles, mais elles donnent tous accès aux processeurs comme espace d'adresse globale.

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.**

- ***Uniform Memory Access** (UMA):*

Est une architecture mémoire partagée utilisée dans les ordinateurs parallèles.Tous les processeurs dans le modèle UMA partagent la mémoire physique, le temps d'accès à la mémoire est Independent du processeur qui la sollicite dans quel boitier de la mémoire se situe la donné à transférer, ainsi chaque processeur peut utiliser son propre cache.

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

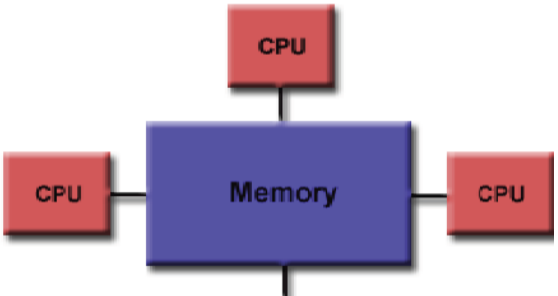


Figure 1.7 Uniform Memory Access (UMA)

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

Signifiant respectivement *accès mémoire non uniforme* et *architecture mémoire non uniforme*) est un système multiprocesseur dans lequel les zones mémoire sont séparées et placées en différents endroits (et sur différents bus). Vis-à-vis de chaque processeur, les temps d'accès diffèrent donc suivant la zone mémoire accédée.

Le système NUMA a été conçu pour pallier aux limites de l'architecture SMP dans laquelle tout l'espace mémoire est accessible par un unique bus engendrant des problèmes d'accès concurrents par les différents processeurs. C'est particulièrement nécessaire pour les systèmes ayant de nombreux processeurs.Si Cache Curency est maintenue ont peut donc l'appeler CC NUMA.

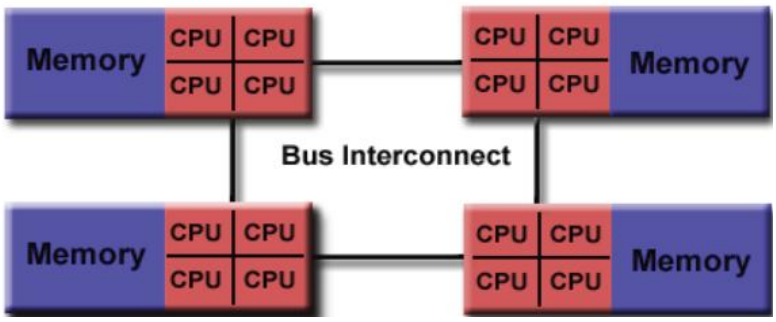


Figure 1.8 Non-Uniform Memory Access (NUMA):

II.1.2 Avantages :

L’espace d'adresse global facilite la programmation.

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

II.1.3 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 couteux avec un grand nombre de processeurs.

II.2. La mémoire distribuée :

II.2.1 caractéristiques générales:

La mémoire distribuée est la mémoire des architectures distribuées, c'est-à-dire la mémoire dite « vive » dans les systèmes informatiques pouvant exécuter en collaboration un programme sur plusieurs machines reliées ensemble par réseau. Or chaque processeur de ces machines, ne peut faire des calculs que sur les données directement contenues dans sa mémoire. Dans ces systèmes parallèles, les machines doivent donc utiliser le réseau pour accéder aux données présentes sur un autre nœud. 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)

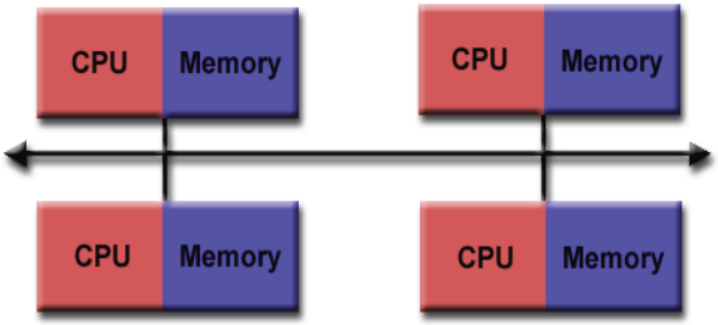


Figure 1.9 mémoire distribuée

Le programmeur doit définir comment et quand les données sont envoyées lorsqu'un processeur à besoin d'accéder aux données d'un autre processeur, et doit encore définir les synchronisations entre les tâches.

II.2.2 Avantages :

L'augmentation du nombre de processeur augmente la mémoire.

Chaque processeur peut accéder directement à ces mémoires.

II.2.3 inconvénients :

Le programmeur est responsable des détails de transmission de données entre les processeurs.

Dans cette organisation mémoire il est difficile d'adresserles structures de données existantes dans la mémoire global.

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

II.3.1 caractéristiques générales

La plupart des ordinateurs dans le monde emploie la mémoire partagée et distribuée ensemble car ceci rend ces ordinateurs plus rapides.

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

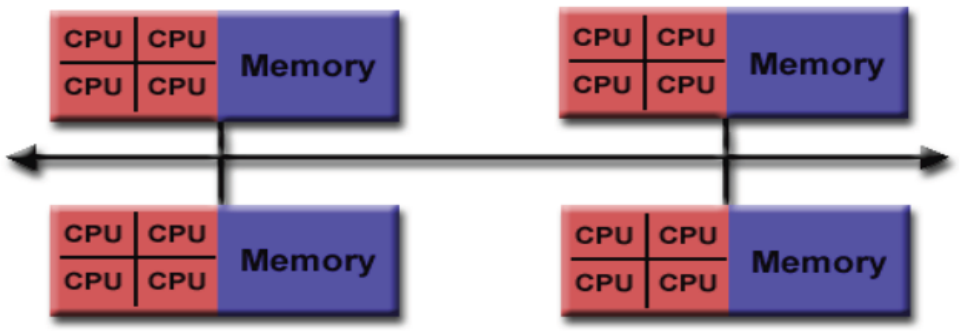


Figure 1.10 La mémoire Hybride (Distribuée-partagée)

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.

III. Les classes du parallélisme :[1]

III.1 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.

III.2 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 parallèlesynchrone 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.

IV. Les modes du parallélisme: [1]

Il est nécessaire de bien comprendre les différences essentielles entre les notions de parallélisme homogène et hétérogène. Au delà des simples concepts, ils constituent deux manières distinctes d'utiliser un certain nombre de processeurs au sein d'un même calculateur. De plus, peu de programmes du domaine scientifique utilisent la programmation hétérogène. Celle-ci est présente essentiellement au niveau système, dans le noyau UNIX, dans les applications temps réel et dans les protocoles de communication par exemple.

IV.1 le parallélisme hétérogène et la programmation concurrente:

Dans ce cas, le parallélisme s'exprime par des tâches exécutant différentes instructions. Ceci consiste à séparer les actions à réaliser en partitionnant les fonctions du programme (function partitioning).

Cette forme du parallélisme est typiquement présente dans ce que nous appellerons la programmation concurrente par opposition à la programmation parallèle. Les algorithmes du type producteur/ consommateur, maitre/esclave, moniteur sont des exemples. Les outils de base qui sont utilisés sont les threads d'exécution, les verrous, les sémaphores et les barrières. Leur utilisation est complexe étant donnée la nature typiquement asynchrone de la programmation concurrente.

L'écriture d'un système d'exploitation tel qu'UNIX fait largement appelle à ces algorithmes, par exemple dans le sequencement (scheluding) des différentes tâches, la gestion des changements du contexte ou la gestion de la mémoire partagée et les synchronisations utilisées par les sémaphores.

IV.2. Parallélisme homogène et programmation parallèle:

Cette forme de parallélisme est basée sur l'application d'une même tâche à des données différentes, c'est donc un parallélisme sur les données et la programmation associée est dite parallèle. La philosophie dans ce cas est de préparer le traitement des données en les partitionnant.

V. Les supports du parallélisme : [2]

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. Pour maintenir la consistance des données, l'accès aux variables partagées 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.

V.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.

VI.1.1. 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.

V.1.2. 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.

V.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.

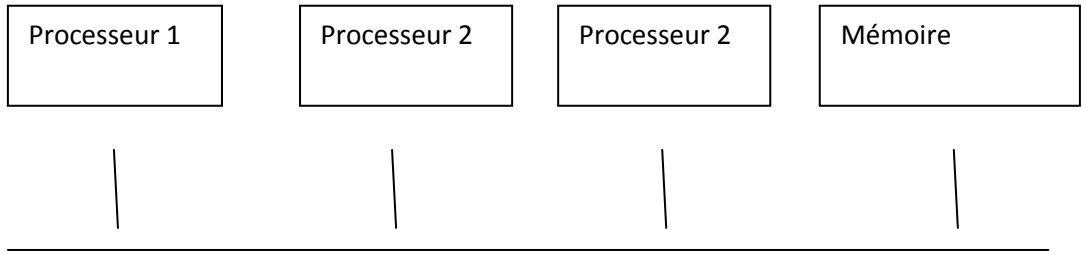


Figure 1.11 partage de la mémoire par plusieurs processeurs sans la cohérence des caches

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

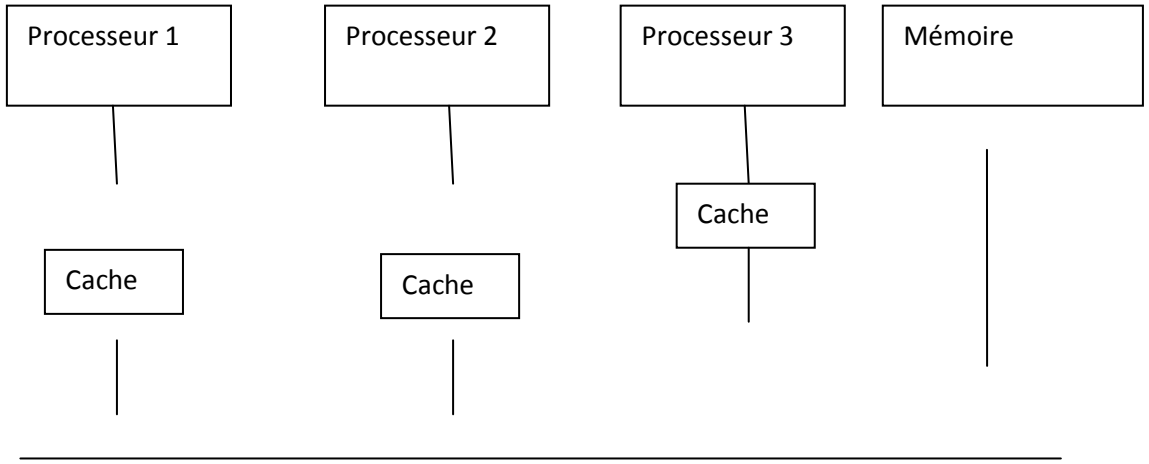


Figure 1.11 partage de la mémoire par plusieurs processeurs avec la cohérence des caches

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.

V.3. La dépendance des données :

Pour qu'une boucle soit parallèlisable, chaque itération doit être indépendante des autres. Un moyen simple de s'en assurer consiste à exécuter la boule dans un sens (indice incrémenté) puis dans l'autre (indice décrémenté) et vérifier que les résultats sont identiques. Une dépendance des données n'intervient que lorsque les caches mémoire sont modifiées, une boucle possède une dépendance de donnée si une itération écrit une variable qui est lu ou écrite dans une autre itération de la boucle. Il n'y a pas de dépendance si une seul itération lit ou écrit une variable en mémoire ou si plusieurs itérations lisent la même variable sans la modifier.

Les dépendances peuvent être de plusieurs types :

- ***Dépendance de contrôle : Instruction IF ou GOTO.***

```
DO  I=1 , N
        If  (T(I)  .LT.  0.)  goto  100
        T(I)  =  T(I) * alpha
END  DO
100  CONTINUE
```

Cette boucle contient un branchement de sortie, le flux de contrôle peut donc sortir de la boucle soudainement. Il n y a pas de moyens simple de paralléliser automatiquement cette boucle.

- ***Anti dépendance***

Une variable est utilisée puis assignée dans le même corps de boucle.

```
Y  = X ;
X  =3 ;
```

- ***Dépendance de sortie :***

La variable est assignée puis réassignée dans le même corps de boucle.

Y = 4

X = 3

- ***Dépendance des flux :***

  - ***Dependence avant (forward dependence).***

Do i = 1 , n

X(I) = X(I+1)

Y(I-1) = Y(I)

End do

X(I) ou Y(I-1) est utilisé pendant une itération et est redéfini dans une itération suivante. X(I) remplacé par la vielle valeur de X(I+1). Il est nécessaire de synchroniser la boucle pour effectuer la bonne valeur avant qu'elle n'ait mise à jour.

  - ***Dependence arrière (backward dependence) :***

Do i = 1, N

X(I) = X(I-1)

Y(I+1) = Y(I)

End do

X(I) ou Y(I+1) est définie pendant une itération et utilise à l'itération suivante, l'écriture doit donc se poursuivre avant la lecture et le seul moyen de paralléliser cette boucle est d'introduire des synchronisations pour garantir l'ordre d'exécution les lectures et écritures. Il n'y a pas de moyens simple de paralléliser cette boucle.

- ***La reduction :***

```
Do I = 1, N
    S = S + A(I)
End  Do
```

L'opération sur S est une réduction, S dépend de sa valeur à une autre itération.

VI. Les modèles de programmation parallèle : [1]

VI.1. Le parallélisme de contrôle :

Cette forme de parallélisme permet l'exécution de deux ou plusieurs opérations simultanément. Ces opérations peuvent être plus ou moins complexes ou plus ou moins indépendantes. Chaque opération est prise en charge par un processus, généralement affecté à un processeur donnée pour la durée de son exécution. Le cas idéal consiste en N actions totalement indépendantes qui s'exécuteront N fois plus vite sur N processeurs que sur un seul.

VI.2. Parallélisme de flux :

C'est le parallélisme du travail à la chaine ou 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 programmation parallèle est égale au nombre d'étages du pipe-line. Il y' a dépendance entre les instructions lorsqu'elles réfèrent au même mot mémoire, l'une au moins de ces références implique une modification de ce mot.

VI.3. Parallélisme sur les données :

Cette forme de parallélisme exploite la structure régulière des données (vecteurs, matrices …) en répétant la même action sur des éléments différents de la structure traitée, les données sont associées à un processeur.

Il est possible d'exploiter ce parallélisme en mode synchrone, mais également en mode asynchrone, la différence principale réside dans l'existence d'une seule instance du programme en SIMD, alors que dans MIMD nécessite une copie du programme dans chaque processeur, conduisant à une grande consommation de mémoire. Afin de pouvoir atteindre un bon niveau d'efficacité, les données doivent être présentes en grande quantité afin de maximiser l'utilisation des appels systèmes ou des dispositifs matériels. En effet seul le processeur qui a la condition vrai du test exécutera les instructions du corps de test alors que les autres, correspondant à la condition fausse ne feront rien et ceci conduit à une perte d'efficacité certaine.

VII. terminologie liée à la programmation parallèle :

Tâche : une section logique discrète d'un travail de calcul. Une tâche est un programme ou un ensemble d'instructions exécutées par un processeur.

Tâche parallèle : une tâche qui peut être exécutée par plusieurs processeurs.

Exécution en série : exécution séquentielle du programme, une instruction à la fois.

Exécution parallèle : exécution de plus d'une tâche du programme, chaque tâche exécute la même instruction ou une instruction différente à la fois.

Symmetric Multi-Processor (SMP) :le Symmetric multiprocessing (SMP) est une architecture informatique dite parallèle qui consiste à multiplier les processeurs au sein d'un ordinateur, de manière à augmenter la puissance de calcul.

L'augmentation du nombre de processeurs permet d'exécuter simultanément un plus grand nombre de processus du système, utilisateur ou noyau en leur allouant l'un ou l'autre des processeurs disponibles.

Hashmap : une table de hachage est en informatique, une structure de données qui permet une association clé-élément, c'est-à-dire une implémentation du type abstrait table de symboles.

On accède à chaque élément de la table via sa clé. Il s'agit d'un tableau ne comportant pas d'ordre (un tableau est indexé par des entiers). L'accès à un élément se fait en transformant la clé en une valeur de

hachage (ou simplement hachage) par l'intermédiaire d'une fonction de hachage. Le hachage est un nombre qui permet la localisation des éléments dans le tableau, typiquement le hachage est l'index de l'élément dans le tableau.

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. Son nom est inspiré du proverbe « Diviser pour régner »

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. Opèrent dans une boucle qui récupère les requêtes de la file de travail (aussi attendre une requête si nécessaire), traitent la requête appropriée et retourne à la liste de travail pour une requête suivante.

Conclusion :

Dans ce chapitre nous avons vue des notions générales sur les ordinateurs parallèles et la programmation parallèle, maintenant on passe au chapitre deux ou on va expliquer d'une façon détaillée les différentes étapes de la conception d'un programme parallèle avec quelques exemples de programmes.

Chapitre 2

La conception des programmes parallèles

Introduction :

La conception et le développement d'un programme parallèle était un processus manuel, le programmeur doit identifier et implémenter lui-même tout le parallélisme, ce développement manuelle prend beaucoup de temps et fait des programmes complexe, exposé à l'erreur et avec un processus itératif.

Notre démarche de conception consiste à comprendre le problème à résoudre, décomposer le problème en plusieurs sous tâches, définir les communications entre les tâches (si elles existent), gérer les dépendances de données (si elles existent) enfin partager le travail entre les tâches.

Actuellement plusieurs outils sont envisagés pour convertir automatiquement un programme séquentiel en un programme parallèle et la plupart de ces outils sont des compilateurs du parallélisme et des préprocesseurs.

On commence le chapitre 2 par une petite description des compilateurs parallèles ensuite on va expliquer d'une façon détaillée le processus de conception d'un programme parallèle.

I. Les compilateurs parallèles :[13]

Les compilateurs parallèles travaillent avec deux façons différentes :

I.1. complètement automatique :

Ces compilateurs analysent le code source et identifient les codes parallèles possibles, ces compilateur fournissent des renseignements sur le comportement et le coût du parallélisme pour voir si le parallélisme améliore la performance. Les boucles sont les plus destinées pour le parallélisme avec ces compilateurs.

I.2. Dirigé par le programmeur :

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 sont présentes, le parallélisme automatique est la bonne solution, mais les inconvénients suivants peuvent se présenter :

- de faux résultats peuvent se produire.

- La dégradation de la 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.

II. Conception d'un programme parallèle:[13]

II.1 comprendre le problème :

La première étape dans le développement d'un logiciel parallèle est de comprendre le problème à résoudre en parallèle, si on commence avec un programme séquentiel on doit d'abord le comprendre.

Avant d'essayer de développer une application parallèle on doit d'abord vérifier si le programme peut être parallèle.

- **Exemple d'un programme parallèlisable :**

  Calculer l'énergie potentielle de chaque molécule ensuite trouver l'énergie minimum.

Ce problème peut être résolu en parallèle, chaque molécule est indépendante et le calcul du minimum est aussi un problème parallèlisable.

- **Exemple d'un programme non parallèlisable :**

  Le calcule de la série de Fibonacci (1,1,2,3,5,8,13,21,...) en utilisant la formule :

  $F(k + 2) = F(k + 1) + F(k)$.

C'est un problème non parallèlisable parce que le calcul des séquences de la série de Fibonacci sont dépendantes, pour calculer le terme K+2 il faut avoir les termes K+1 et K.

- **Identification des goulots du programme :**

  Voir s'il existe des partie du code qui ralentissent le programme comme les entrées/ sorties.

Utiliser plusieurs algorithmes ou structurer le programme avec de différentes façons pour réduire ou éliminer les parties du code qui ne sont pas nécessaires et qui ralentissent l'exécution du programme

- **Identification des hotspots du programme :**

Les hotspots sont des ressources (code, mémoire …) où se déroule le travail important du programme. Pour cela il faut se concentrer sur les **hotspots** et ignorer les sections du programme qui utilisent peut de ressources.

Les outils d'analyse peuvent bien aider à mieux connaitre le comportement du programme.

II.2. Le partitionnement ou décomposition :

Une des premières étapes dans la conception des programmes parallèles est de décomposer le problème en plusieurs parties discrètes qui peuvent être attribuées aux tâches.

Une bonne décomposition divise le programme en petites parties, que se soit pour le traitement ou les données associées. A cette phase de conception les programmeurs doivent se concentrer sur les données associées au problème. Déterminer une répartition appropriée pour les

données ensuite associer les traitements pour ces données (décomposition du domaine), décomposer le traitement ensuite associer les données (décomposition fonctionnelle).

Il existe deux façons pour décomposer un calcul en plusieurs tâches : la décomposition du domaine et la décomposition fonctionnelle.

II.2.1 La décomposition du domaine :

Dans ce type de décomposition, on détermine une répartition appropriée pour les données ensuite on associe les traitements pour ces données, de préférence diviser ces données en plusieurs pièces identiques

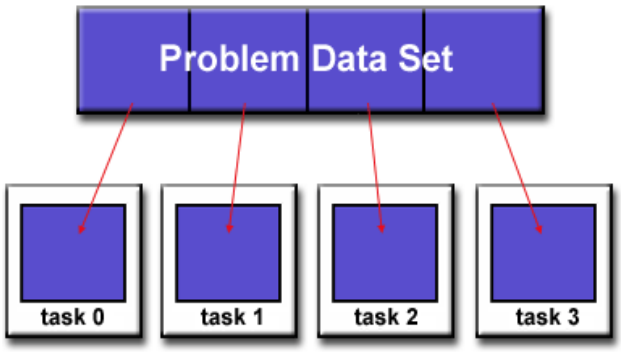


Figure 2.1 décomposition du domaine

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

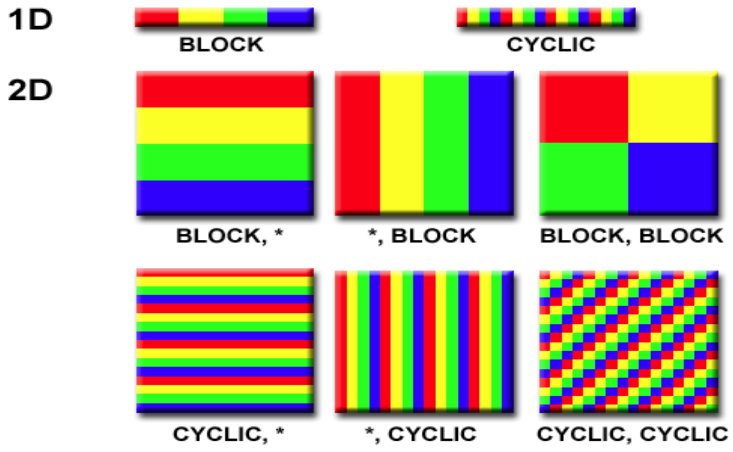


Figure 2.2 décomposition par bloc etdécomposition cyclique des données

II.2.2 La décomposition fonctionnelle :

Dans cette approche, on se base sur le traitement qui va s'exécuter plutôt que les données manipulées par ce traitement. Le problème set décomposé en fonction du travail à faire, chaque tâche exécute une partie du travail.

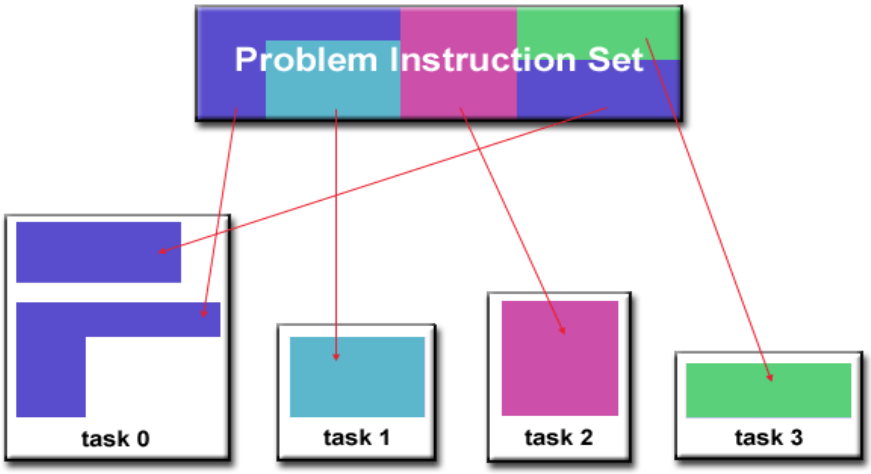


Figure 2.3 décomposition fonctionnelle

II.2.3 Exemples :

- **modélisation d'un système écologique :**

Chaque programme traite la population d'un groupe donné, ou chaque groupe dépend de ces voisins, quand le temps passe chaque processus calcule son état courant ensuite change l'information avec les populations voisines. Toutes les tâches progressent pour calculer l'état de la prochaine étape.

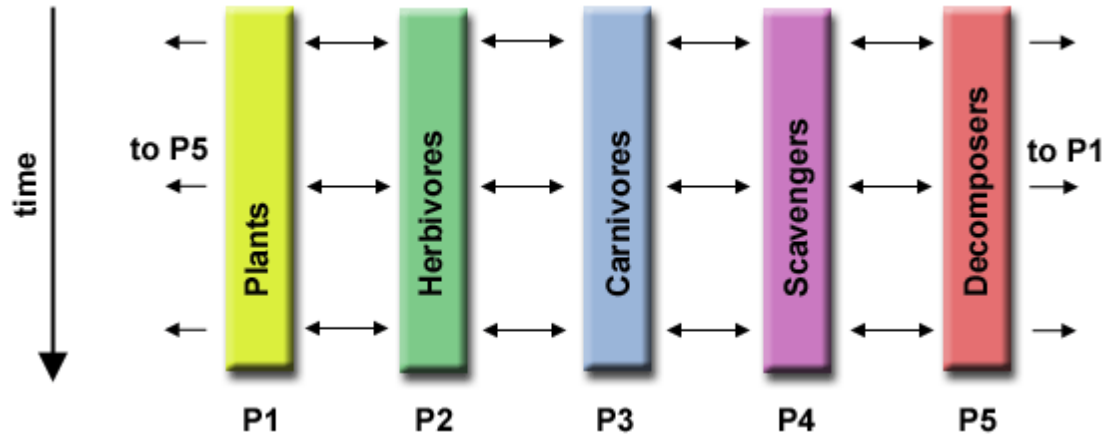


Figure 2.4 modélisation d'un système écologique

- **L'échange des données :**

Les données sont passées par quatre filtres distincts, chaque filtre est un processus à part. Le premier segment de données doit passer par le premier filtre avant de passer aux filtres suivants, une fois passé au deuxième filtre, un deuxième segment de données peut passer par le premier filtre, et ainsi de suite. A la fin toutes les quatre tâches seront occupées.

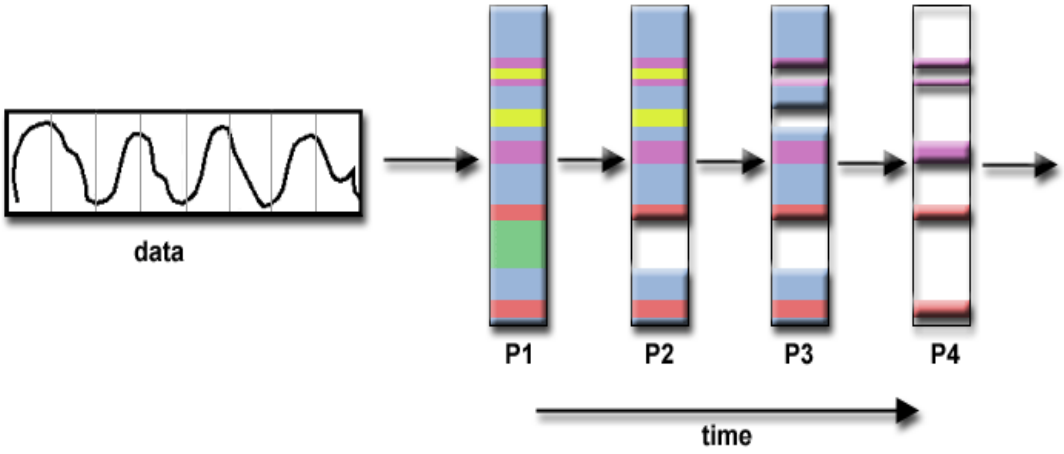


Figure 2.5 échange des données

- **La modélisation du climat :**

Chaque composant du modèle peut être considéré comme une tâche séparée, les flèches représentent les échanges de données entre les composants pendant le traitement : le modèle atmosphérique génère les données de la vitesse du vent qui seront utilisées par le modèle océan, le modèle océan génère les données qui concernent la température de la surface modèle qui seront utilisées par le modèle atmosphère et ainsi de suite.

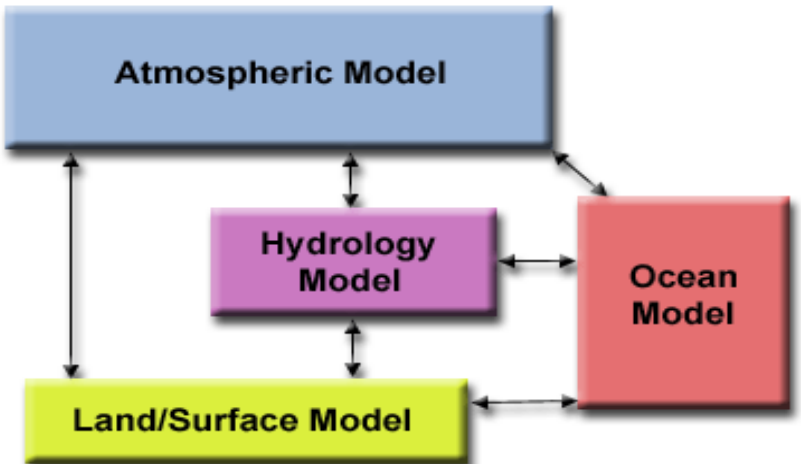


Figure 2.6 modélisation du climat

On note aussi qu'on peut combiner ces deux types de décomposition.

II.3. Les communications :[12]

Quand une tâche partage des données avec d'autres tâches, les données doivent être transférées entre les tâches pour permettre l'exécution du traitement. Ce flux d'informations constitue la phase de communication dans la conception.

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

Dans un problème à décomposition de domaine, les communications sont difficiles à déterminer. Dans cette stratégie de décomposition, quelques opérations ont besoin de plusieurs tâches, donc la communication est indispensable pour gérer le transfert de données. Organiser la communication d'une façon efficace n'est pas facile, même une simple décomposition peut avoir une structure de communication complexe.

Cependant avec la décomposition fonctionnelle la communication est plus au moins facile à définir car elle constitue un flux de données entre les tâches. Dans l'exemple précédent (modélisation du climat) la communication correspond aux interfaces entre les sous modèles.

Le besoin à la communication entre les tâches dépend du problème résoudre.

- **Les programmes qui n'ont pas besoin de la communication :**

Quelques problèmes peuvent être décomposés et exécutés en parallèle et leurs tâches n'ont pas besoin d'échanger les données, par exemple une opération qui détecte les couleurs d'une image, les données de l'image peuvent être distribuées à des tâches différentes qui traite une portion de l'image indépendamment des autres.

- **Les programmes qui nécessitent de la communication :**

Quelques applications nécessitent que leurs tâches partagent les données entre elles, dans un problème de diffusion de chaleur une tâche doit savoir la température de sa voisine et le changement des données d'une tâche implique le changement des données de sa voisine.

II.3.1 La communication locale :

Dans une structure de communication locale, une opération récupère des données par un nombre restreint de tâches. Dans ce cas, le canal de communication, les tâches liées à ce canal (producteurs et consommateurs), et les opérations qui se chargent d'envoie et de réception des données sont faciles à définir.

II.3.2 La communication globale :

Une opération de communication globale est une opération où un grand nombre de tâches participent, quand on implémente ce type de communication, il ne suffit pas de définir les couples producteur/consommateur, dans ce cas il faut mettre une stratégie de communication permettant une exécution concurrente.

II.3.3 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. La communication synchrone utilise généralement l'approche de communication avec blocage.Elle peut être définit d'une façon explicite dans le code du programmeur et elle peut aussi être implémentée dans le niveau bas du hardware.

II.3.4 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 producteurs. 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. La communication asynchrone utilise généralement l'approche de communication sans blocage puisque le reste du travail peut se faire sans le besoin de la terminaison de la communication.

II.4 La dépendance des données :

La dépendance existe dans les parties du programme quand le changement d'ordre de ces parties affects les résultats de ce programme. La dépendance des données se produit quand plusieurs tâches utilisent la même localisation de sauvegarde.

II.4.1 Exemple de données dépendantes :

- **Exemple 1 :**

```
DO 500 J = MYSTART,MYEND

        A(J) = A(J-1) * 2.0

500 CONTINUE
```

La valeur de A(J-1) doit être calculée avant A(J), donc A(J) récupère des données de A(J-1).

Si la tâche 1 contient a(j-1) et la tâche 2 contient a(j), le calcul de a(j) nécessite :

La valeur de a(j-1) de la tâche 1 après que la tâche 1 finit son calcul, dans une architecture à mémoire distribuée. La lecture de a(j-1) après que la tâche 1 finit son calcul, dans une architecture à mémoire partagée.

- **Exemple 2 :**

Tache1	**tache2**
X = 2	*X = 4*
.	.

. .

*Y = X**2* *Y = X**3*

Dans cet exemple la valeur de Y dépend de :

- La valeur de X qui est communiquée entre les taches dans une architecture à mémoire distribuée.
- La dernière valeur enregistrée par la dernière tache dans une architecture à mémoire partagée.

II.4.2 La gestion des dépendances des données :

Dans une architecture à mémoire distribuées : la communication des données dans des points de synchronisation.

Dans une architecture à mémoire partagées : synchroniser les opérations lire/écrire entre les tâches.

II.5 La répartition des tâches (Load Balancing) :

La répartition des tâches c'est le partage du travail entre les tâche de façon que toutes les tâches soient en exécution pendant tout le temps, La répartition des tâches est très importante pour les programmes parallèles pour des raisons de performance, par exemple si toutes les tâches arrivent à un point de synchronisation, c'est la tâche la plus lente qui détermine la performance du programme.

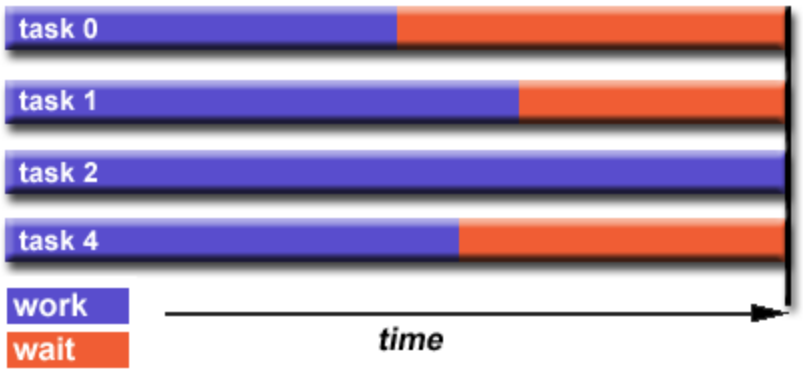


Figure 2.7 La répartition des tâches (Load Balancing)

II.5.1 Répartition statique de travail entre les tâches :

Pour les opérations liées aux tableaux et aux matrices ou chaque tâche exécute le même travail, on distribue les données aux tâches.

Pour les boucles qui exécutent les mêmes instructions à chaque itération, on distribue les itérations aux tâches.

II.5.2 Répartition de travail dynamique :

Certains problèmes restent présents même si on distribue les données aux tâches :

Les tableaux vides : quelques tâches ont les données actuelles pour exécuter leurs tâches mais d'autres n'ont pas.

Les méthodes adaptées : quelques tâches doivent fournir leurs travail aux autre tandis que d'autre n'ont pas besoin de le faire.

Utiliser l'approche «pool d'ordonnancement des tâches » : quand une tâche finit son travail elle sera empilée dans une file pour lui attribuée un autre travail à accomplir.

Il est aussi nécessaire de définir un algorithme qui détecte et qui gère les Load Imbalances (le travail qui ne peut pas être partagé d'une façon égale entre les tâches).

III. Exemples de conception de programmes parallèles : [13]

III.1 Opérations sur les tableaux :

III.1.1 Description du problème :

Dans cet exemple on prend un tableau à deux démentions et on essaye de manipuler les éléments de ce tableau sachant que ces éléments sont indépendants les uns les 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

III.1.2 Solution parallèle 1: partage statique du travail.

- **partage du travail:**

Le tableau est partagé en plusieurs portions identiques et chaque portion 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 if I am WORKER
        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 2.8 solution parallèle 1 :partage 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.

III.1.3 Solution parallèle 2 : pool de tâche

La solution précédente utilise le partage statique du travail, chaque tâche a 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.

- **Partage de travail : dynamique**

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 if I am WORKER

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

endi

Figure 2.9Solution parallèle 2 : pool de tâche (Opérations sur les tableaux)

III.2 Calcule de PI

III.2.1 Description du problème :

Le calcul de PI peut se faire avec plusieurs méthodes, on considère la méthode suivante qui calcule approximativement le nombre PI :

1. Définir un cercle dans un carré.
2. Générer d'une façon aléatoire des points dans le carré.
3. Déterminer le nombre des points dans le carré qui sont aussi dans le cercle.
4. Mettant r le nombre des points dans le cercle divisé par le nombre total des points.
5. PI ~ 4*r

La solution séquentielle de ce problème est la suivante :

```
npoints = 10000

circle_count = 0


do j = 1,npoints

        generate 2 random numbers between 0 and 1

        xcoordinate = random1

        ycoordinate = random2

        if (xcoordinate, ycoordinate) inside circle

        then circle_count = circle_count + 1

end do


PI = 4.0*circle_count/npoints
```

Figure 2.10 solution séquentielle (Calcule de PI)

On note que la plus part de temps d'exécution est consommé par la boucle.

III.2.2 La solution parallèle :

- **Le partage du travail :**

Diviser la boucle en plusieurs portions afin de les partager sur différentes tâches.

Chaque tâche exécute sa partie de la boucle qui lui est affectée.

npoints = 10000

circle_count = 0

p = number of tasks

num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num

generate 2 random numbers between 0 and 1

xcoordinate = random1

ycoordinate = random2

if (xcoordinate, ycoordinate) inside circle

then circle_count = circle_count + 1

end do

if I am MASTER

receive from WORKERS their circle_counts

compute PI (use MASTER and WORKER calculations)

else if I am WORKER

send to MASTER circle_count

endif

Figure 2.11 solution parallèle (Calcule de PI)

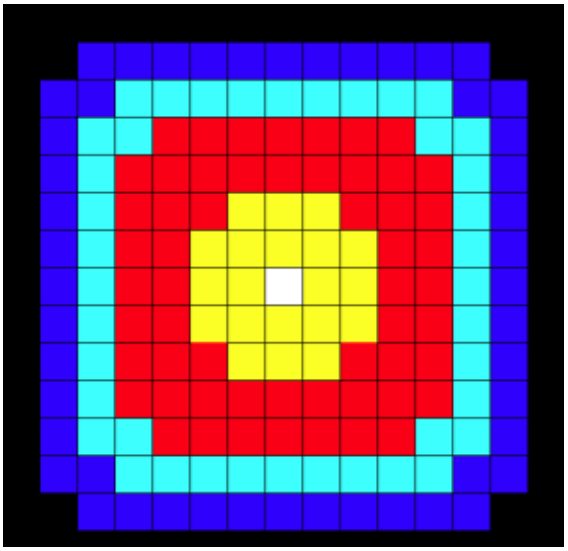
Dans cet exemple les tâches peuvent faire leurs travail sans le besoin d'informations des autres tâches (il n'y a pas de dépendance de données).

III.3 Une équation qui calcule la chaleur

III.3.1 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 2.12 calcule de la chaleur



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

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

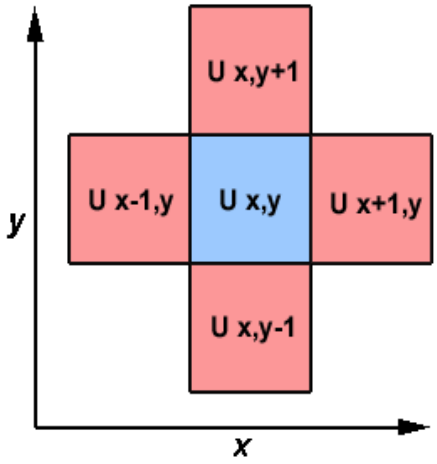


Figure 2.13 équation qui calcule la chaleur

Le programme séquentiel 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
```

Figure 2.14 solution séquentielle (Calcule de la chaleur)

III.3.2 Solution parallèle 1:

- **Le partage du travail :**

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

- **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 dépendent 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.

- **La communication entre les tâches : approche avec blocage.**

Pour les éléments intérieurs il n'y 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 :

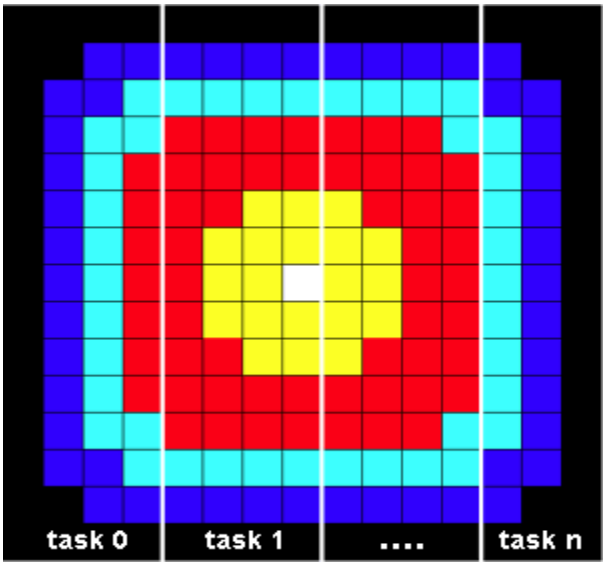


Figure 2.15 Solution parallèle 1(Calcule de la chaleur)

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 if I am WORKER
    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 2.16 Solution parallèle 1:approche avec blocage(Calcule de la chaleur)

III.3.3 solution parallèle 2 (utilisation des communications sans blocage)

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

- **La communication entre les tâches : approche sans blocage**

En utilisant une approche de communication sans blocage on peut réduire le temps d'attente des processus nécessitant la communication. L'approche sans blocage 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 if I am WORKER
        receive from MASTER starting info and subarray

        do until solution converged
        update time
```

Figure 2.17Solution parallèle 2:approche sans blocage(Calcule de la chaleur)

```
                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 2.17 (suite)Solution parallèle 2:approche sans blocage(Calcule de la chaleur)

Conclusion

Dans ce chapitre nous avons étudié le processus de conception d'un programme parallèle, de la première étape qui consiste à comprendre le programme jusqu'au partage du travail entre les tâches en passant par les communications et les dépendances de données. Les exemples à la fin du chapitre nous ont permis de mieux assimiler ce processus de conception en l'appliquant sur des exemples concrets.

Cependant le processus de conception d'un programme parallèle devient plus délicat pour les problèmes de grande de taille qui utilisent des grandes quantités de données et des ressources et qui peuvent être aussi réparties sur plusieurs sites distants, pour ces types de programmes, le programmeur trouve souvent des difficultés dans la gestion du parallélisme. Pour cela des outils dédiés pour la mise en œuvre des programmes parallèles ont été envisagés. Ces outils permettent d'implémenter le parallélisme en utilisant des mots clés simples.

Le chapitre suivant traite les outils permettant d'implémenter des applications desktopparallèles.

Chapitre 3

Outils pour le parallélisme dans les applications desktop

Introduction

Plusieurs solutions ont été envisagées pour introduire l'ingénierie du logiciel Multicore pour les applications desktop, les chercheurs ont même définirent les frontières de leurs recherches pour le développement future. On commence le chapitre 3 par une description des caractéristiques des applications desktop, ensuite on proposera quatre outils pour le parallélisme dans les applications desktop.

IntelTBB (Intel threading building blocks), un bon exemple d'une bibliothèque parallèle. Cette bibliothèque réduit d'une façon remarquable la durée du développement des applications desktop, nous allons expliquer la conception et le fonctionnement interne de cette bibliothèque ainsi la façon de son utilisation.

Dans “l'approche de refactoring pour le parallélisme” on propose une démarche à suivre pour produire un programme parallèle à partir d'un programme séquentiel déjà existant, on a introduit les différentes catégories de refactoring en java: sécurité des transformations, la vitesse d’exécution, la

portabilité et nous allons par suite présenter un outil qui aide le programmeur de construire son programme assez vite.

Qt C++ et ParaTask de java deux bibliothèques parallèles qui utilisent les concepts du parallélisme dans un environnement orienté objet

I. Caractéristiques d'une application desktop :[7]

La figure 3.1 illustre un exemple d'application desktop, l'utilisateur s'interagisse avec l'application

pour effectuer les différentes tâches. Quelque taches s'exécutent une seul fois tandis que d'autres s'exécutent plusieurs fois en utilisant des données différentes (figure 3.1-a), quelques taches exécutent des calcules intensifs tandis que d'autres se limitent à des entrées/ sorties telque recherche sur internet (figure 3.1-b), en attendant la demande de l'utilisateur (figure 3.1-c) ou encor imprimer (figure 1d). Quelques tache s'exécutent indépendamment tandis que d'autres doivent respecter un ordre d'exécution (figure 3.1-a). On retrouve plusieurs types de taches avec des comportements différents qui demandent des traitements différents.

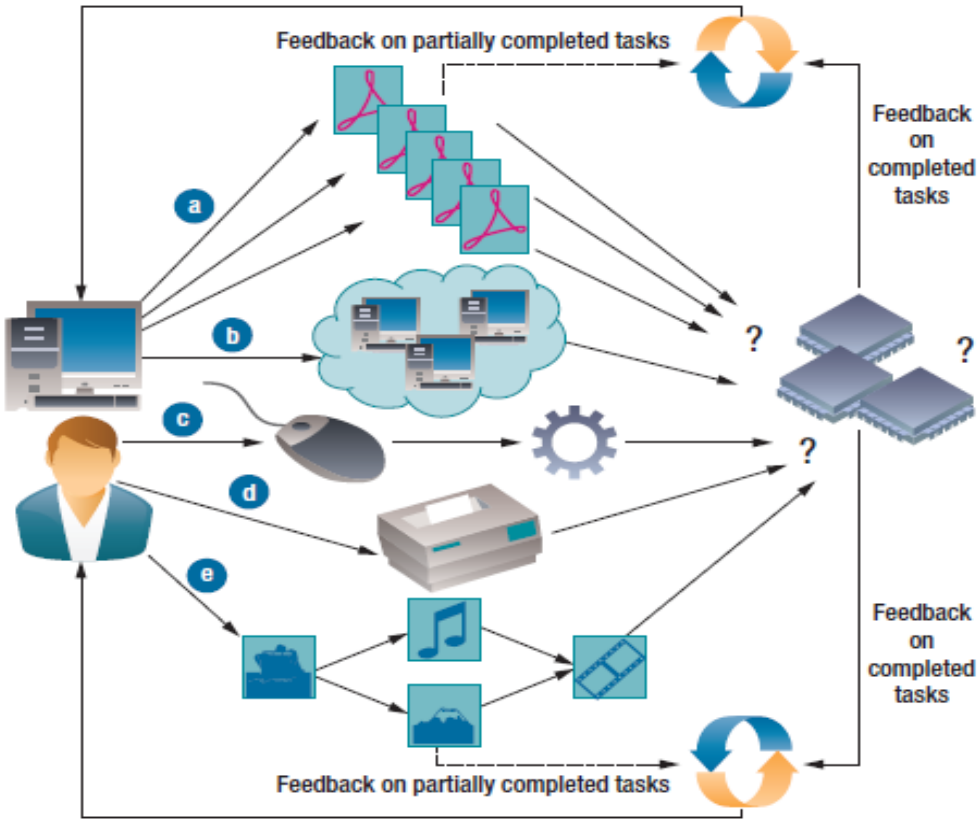


Figure 3.1 caractéristiques d'une application desktop

Avant de penser au parallélisme nous devons comprendre le comportement interne et externe des applications desktop. Les caractéristiques externes nous semblent plus familières telque boutons, labels, champs de texte…

Pour la structure interne c'est un peu plus compliqué, le plus vital est la boucle d'événement (event loop figure 3.2-a), qui détecte les événements comme le clique de la sourie pour les associer à des événements handler appropriés (figure 3.2-b).

Le thread GUI appelé aussi « Event Despatch Thread» (EDT) se charger des événements qui viennent des composants externes pour effectuer des traitements internes, aucun autre thread 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 monoprocesseur, les threads partagent le processeur dans le temps, le calcule est passé à un autre thread (Tread Helper - figure 3.2-c) et le thread GUI retourne à la boucle d'événement ainsi les deux threads restent actifs, quoique le thread helper exécute le calcule (figure 3.2-d), il n y a aucun accès directe aux composant

(figure 3.2-e) car sauf le thread GUI qui est responsable des composants GUI, ainsi les événements doivent être capturer par le thread GUI (figure 3.2-g) et gérés par le thread helper(figure 3.2-f) .

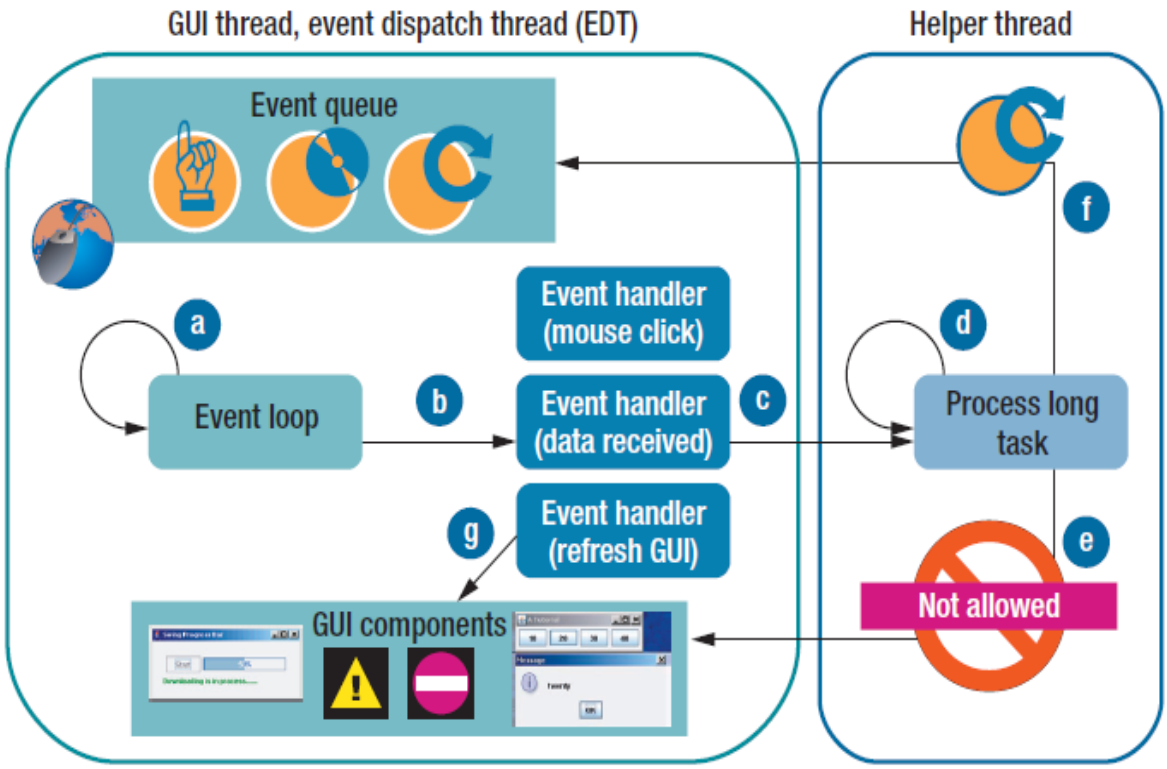


Figure 3.2 Structure du multithreading dans les applications desktop.

La figure inclue : (a) la boucle d'événements, (b) event handler (gérant d'événement), (c) un autre thread, (d) traitement chargé à un autre thread, (composants GUI), (f) événements GUI et (g) thread GUI.

Pour les systèmes à multiprocesseurs on utilise les threads non seulement pour la programmation concurrente mais aussi pour un parallélisme réel, malheureusement le concept de thread gère mal les diverse demande de la programmation parallèle, pour décharger un calcule pour un autre thread il faut diviser et distribuer le calcule pour les différents threads affin d'occuper tout les processeurs. Le problème dans la gestion des threads se pose lorsque les synchronisations sont nécessaires entre les taches pour assurer des résultats correctes.

II. Intel Threading Building Blocks [5]

II.1 Introduction:

Les caractéristiques des applications et de l'environnement compliquent le développement des applications desktop parallèles. Une application desktop est un ensemble de modules qui se dépendent entre eux et qui partages les mêmes ressources, et quand les modules ne sont pas bien composés chaque composant tente de consommer tout les ressources.

Les applications desktops parallèles doivent adapter aux changements de la disponibilité des ressources, ainsi écrire un programme parallèle est une tache difficile et écrire un programme parallèle modulaire qui s'exécute dans un environnement parallèle est une tache plus difficile.

Intel a proposer une bibliothèque de Template C++ qui s'appelle Intel threading building blocks (Intel TBB) affin d'aider les développeurs à mieux écrire des applications parallèles modulaires dans un environnement parallèle.

II.2 À propos de Intel TBB :

Intel TBB est un composant de Intel PBB (Intel parallel building blocks) qui inclus aussi Intel Cilk Plus et Intel Array Building Blocks, Intel TBB est disponible comme un produit commercial et open source et s'exécute dans des plateformes différentes telles que Windows, Linux et Mac OS.

II.3 L'interface de taches :

La souplesse des tache et la notion work-stealing pour la planification des taches sont les clés de performance de IntelTBB, le programmeur exprime les unités parallèle comme étant des taches qui sont des objets C++, et puisque l'allocation des taches est moins léger que l'allocation des threads, le programmeur peux sur décomposer son problème et créer plus de taches que les threads hardware, Intel TBB affecte les ressources pour cet ensemble de taches.

II.4 La notion de Work-stealing:

Intel TBB utilise la notion de work-stealing pour distribuer les taches aux threads worker, par défaut Intel TBB crée un thread worker par thread hardware, chaque thread worker maintient les deux extrémités de sa liste des taches. Quand une tache est engendrée, cette tache sera mise dans la queue de la liste de taches d'un thread worker. Quand un thread worker finis d'exécuter une tache il prend une autre tache de la queue de la liste.

Si un thread worker ne trouve pas de taches dans sa liste, il prend une tache de la tète d'une liste d'un autre thread worker au hasard, ce qu'on appelle le work-stealing.

Puisque les taches peuvent s'exécutés dans n'importe quel thread worker, IntelTBB peut créer autan de threads qu'il veut selon la disponibilité des ressources.

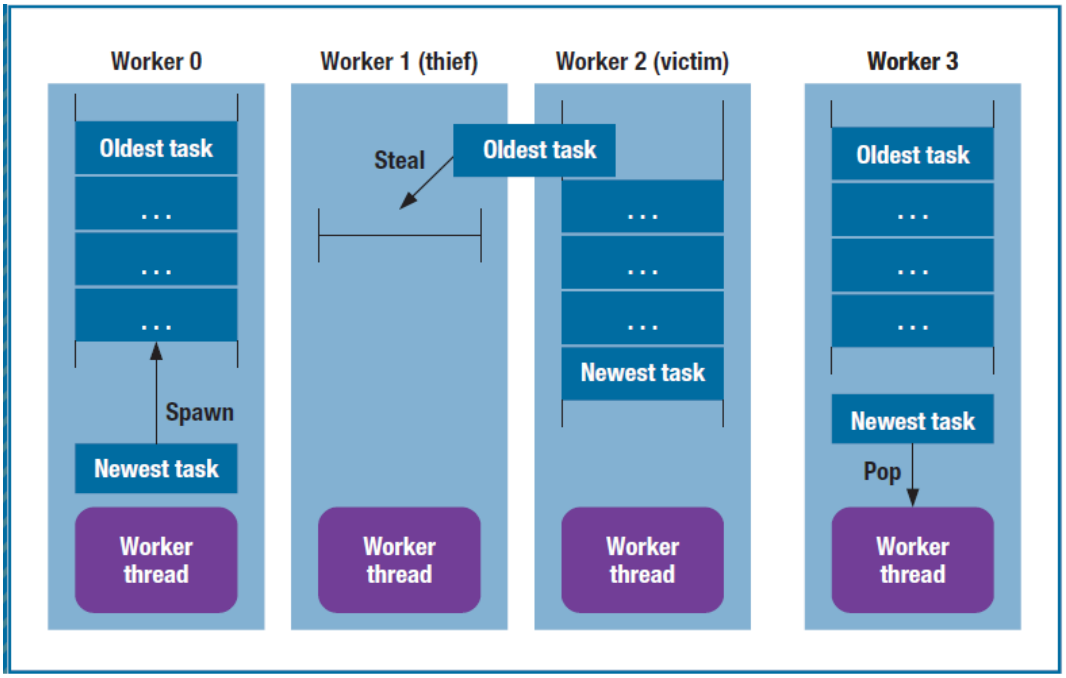


Figure 3.3 le work stealing

II.5 Les composant de IntelTBB :

La figure 3.4 montre les composants (blocks) fournie par IntelTBB pour aider le développeur dans la construction des programmes parallèles :

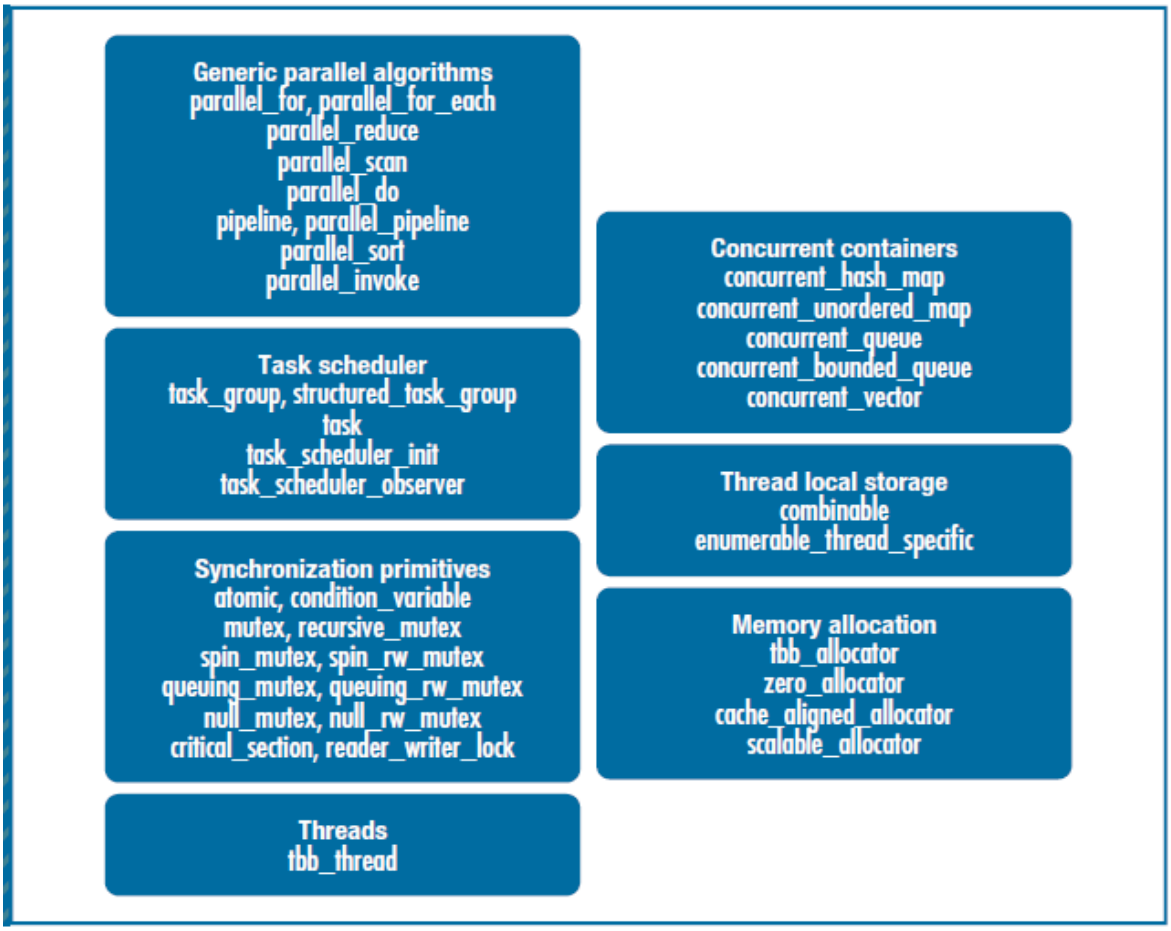


Figure 3.4 : composants de IntelTBB

II.5.1 Utilisation des taches :

IntellTBB fournie deux interfaces pour utiliser les taches : ***task-group***et ***task***. L'interface***task-groupe***est la plus facile à utiliser mais moins flexible. L'exemple suivant (figure 3.5) permet de trouver le minimum dans un arbre binaire en utilisant ***task-groupe***.

La classe task-group définie une fonction **run** et à chaque appelle de cette fonction crée une tache qui exécute se qui est passé comme arguments. Dans cet exemple, quand le nombre de nœuds passe 1000, le programme crée deux taches filles pour parcourir les deus sous-arbres gauche et droite, l'appelle de **g.wait()** bloque le programme jusqu'à ce que toutes les taches dans les groupes soient finis, quand les deux sous arbres se terminent, on obtient le résultat en calculant le minimum des valeurs calculées dans chaque sous arbre.

```
float find_tree_min( tree_node *my_node ) {
  float my_min = FLT_MAX;
  if ( my_node->num_nodes_below > 1000 ) {
     tbb::task_group g;
     float min_left = FLT_MAX, min_right = FLT_MAX;
     if ( my_node->left_child )
        g.run( [&] { min_left = find_tree_min( my_node->left_child ); } );
     if ( my_node->right_child )
        g.run( [&] { min_right = find_tree_min( my_node->right_child ); } );
     g.wait();
     my_min = std::min( my_node->value, std::min( min_left, min_right ) );
  } else {
     my_min = serial_tree_min( my_node );
  }
  return my_min;
}
```

*Figure 3.5 Utilisation de l'interface **task-group** pour calculer le maximum dans un arbre binaire*

Si vous voulez plus de contrôle vous pouvez utiliser **task** à la place de **task_groupe.** L'API **task** permet au programmeur de contrôler le comportement bas niveau de tel que l'affectation des taches pour les threads, et la gestion des exceptions, L'API **task** peux aussi exprimer des dépendances complexe entre les tache,

II.5.2 Les algorithmes parallèles génériques :

IntelTBB fournies des packages d'algorithmes générique construits pour facilité l'utilisation des taches.

II.5.2.1 Itération sur une collection d'objets :

L'utilisateur spécifieune collection d'objet et le corps (traitement) à appliquer pour cette collection, la bibliothèque crée les taches à travers la collection par des subdivisions récursives, comme montré dans l'exemple précédant, ou on a spécifié une collection de 1000 nœuds et le traitement qui est le calcule du minimum de cette collection.

Au premier lieu, la collection sera divisée en deux taches, ensuite la tache décide si elle va encore subdiviser la collection en deux autre taches ou appliquer les instructions appropriées à la collection.

Le schéma suivant (figure 3.6) montre une division récursive qui n'a pas encore déterminer toutes ces sous collections. A gauche le la figure (a), la cases sous forme [0,x) représentent les taches qui sont exécutées ou qui choisissent de subdiviser la collection, et les case sous formes [x1, x2) représentent les taches qui ne sont pas encore exécutées. A droite (b) un schéma illustrant les subdivisions successifs de la collection.

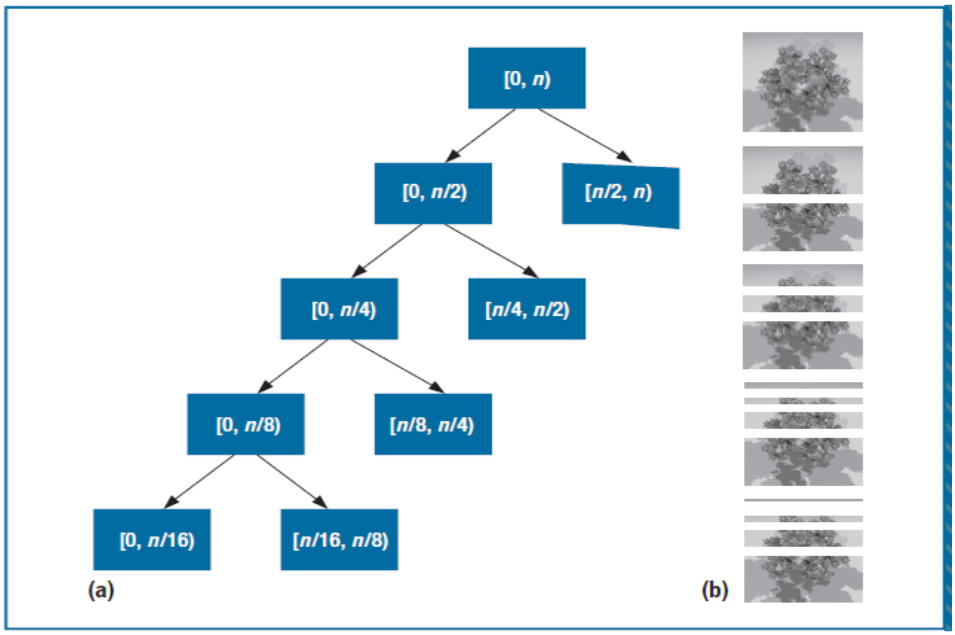


Figure 3.6 Divisions successifs d'une collection pour générer les taches.

Afin de contrôler la terminaison de la récursivité IntelTBB fournie trois politiques de répartition :

- ***Simple-partitioner*** : divise le rang s'il est plus grand du seuil défini par l'utilisateur.
- ***Auto-partitioner*** : par défaut, s'il n y a pas de tache à prendre des autre listes (work-stealing), divise le rang en p*4 sous rang, ou p est le nombre de thread hardware, cependant si une tache est prise (volée) , le thread qui prend cette tache divise cette tache en p*4 sous taches additionnelles.
- ***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.

L'exemple suivant (figure 3.7) est un exemple de l'algorithme **parallel-for** qui montre une répartition des taches en appliquant la politique ***auto-partitioner***

```
class my_body {
float *input;
float *output;
const int N;

public:

my_body( float *in, float *out, int n ) :
input(in), output(out), N(n) {}

void operator()( int i ) const {
output[i] = (input[i-1]+input[i]+
input[i+1])/3;
`
```

```
void DoParallelAverage(float *input,
float *output, int N) {
my_body b( input, output, N );
parallel_for( 1, N-1, b );
}
```

*Figure 3.7 : Exemple d'utilisation de la classe **parallel-for***

*Figure 3.7 Algorithme **parallel_for***

Dans cet exemple, la fonction **DoParallelAverage** appelle le constructeur de la classe **my_body**, qui initialise les variables input, output et N, qui sont respectivement le tableau de sortie, tableau d'entrée et le nombre d'éléments des deux tableaux. Ensuite appelle la classe **parallel_for** en précisant la collection (de 1 à N) et le corps à appliquer pour la collection(p), enfin appelle la classe **operator()** qui maintient output[i] à la moyenne de input[i], input[i-1] et input[i+1] pour 1<= i < n.

La bibliothèque fournit aussi les algorithmes ***parallel-reduce***et ***parallel-scan***qui appliquentun corps d'instruction à une collection d'éléments d'une façon concurrente.

II.5.3 les containers concurrents:

On utilise les algorithmes parallèles que nous avons décrit précédemment pour exprimer le parallélisme dans les applications, ainsi que pour les containers conçurent, les threads utilisent les

containers concurrents dans les applications desktop parallèle pour la synchronisation et la communication.

Un centenaire standard de C++ déjà disponible, mais il n'assure pas la synchronisation entre les threads, ainsi c'est au programmeur d'assurer la synchronisation en utilisant les verrous (mutex).

IntelTBB offre trois types de centenaires concurrents:

- Les containers concurrents associatifs**:*concurrent_hash_ map*, *concurrent_unordered_map***.
- Les vecteurs concurrents**:*concur- rent_vector***.
- Les files concurrentes**:*concurrent_queue, concur_rent_bounded_queue***.

II.5.3.1 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. Sert aux accès concurrents en utilisant des objets appelés (accessors).

Un **accessor** est un pointeur qui se pointe sur la table de hachage, prend un verrous provisoire jusqu'a ce qu'il soit détruit, si la table contient le pair (key, value) qui correspond à la clé d'entrée, les mots clé ***insert*** et ***find*** retournent le pair pour l'accessor et le mot clé ***erase*** détruit le pair de la table.

Cependant la classe ***concurrent_hash_ map,*** à un inconvénient pour les applications qui utilise la table pour la lecture seule, un temps perdu pour les accessors dans chaque verrouillage.

Cet exemple (figure 3.8) montre comment insérer les pair (clé, valeur) avec l'instruction (***htable.insert(a,*p)***), supprimer un pair dans la table de hachage **htable** (***htable.erase(ca)***)**,** et libèrer le verrou (***a.release()***).

```
using namespace tbb;
typedef
 concurrent_hash_map<string,int> StringTableH;
StringTableH htable;
StringTableH::accessor a;
for(string* p=range.begin();p!=range.end();++p)
{
 htable.insert( a, *p );
 a->second += 1;
 a.release();
}

StringTableH::const_accessor ca;
bool b = htable. find( ca, str_key )
        && htable.erase( ca );
```

Figure 3.8 utilisation de la table de hachage ***concurrent_hash_map***

II.5.3.2 Les vecteurs concurrents:

La classe **concurrent_vector** est un tableau d'éléments qui permet l'accès conçurent ainsi une croissance dynamique des éléments. 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 alloué plus d'une seul fois.

L'exemple suivant (figure 3.9) 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) );
```

Figure 3.9 exemple d'utilisation de ***concurrent_vector***

II.5.3.3 Les files concurrentes:

La classe ***concurent_queue***est une structure de donnée de type FIFO, qui permets 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 méthode push permet l'insertion dans la fille, et puisque la taille de la file n'est pas fixée la méthode push renvoie toujours le résultat vraie (si l'espace mémoire est suffisant). La méthode try-pop() récupère un item de la file s'il est disponible.

II.5.4 Les primitives de synchronisation:

De plus des algorithmes parallèles de haut niveau, InteTBB fournies d'autres composants de bas niveau tel que les verrous, et les opérations atomiques.

II.5.4.1 Exclusion mutuelle et verrous:

Les verrous dans IntelTBB sont faciles à utilisés et permettes 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 boucler en attendant que le verrou soit disponible. Dans l'exemple suivant (figure 3.10)le constructeur ***scoped_lokc***attendjusqu'a ce qu'aucun verrous ne reste dans ***spin_mutex***, l'instruction ***spin_mutex::scoped_lock lock(FreeListMutex);*** essaye d'acquérir le verrou, si le verrou n'est pas disponible (***!n***) le thread se boucle (***n=new node()***). À la fin le destructeur restitue le verrou.

```
Node* FreeList; spin_mutex FreeListMutex;
 Node* AllocateNode()
{
        Node* n;
        {
                spin_mutex::scoped_lock
lock(FreeListMutex);
                n = FreeList;
```

Figure 3.10 : exemple d'utilisation de la classe ***spin_mutex***

II.5.4.2 Les opérations atomiques:

Des opérations de bas niveau, utilisés dans les programmes compliqués, IntelTBB fournie cinq opérations fondamental décrites dans le tableau suivant:

Opération	Déscription
= *x*	Read x.
x =	Write to x and return it.
x.fetch_and_store(y)	Do y = x, and return the old value of x.
x.fetch_and_add(y)	Do x += y, and return the old value of x.
x.compare_and_swap(y,z)	If x = z, then do x = y; in either case, return the old value of x.

Figure 3.11 opérations atomiques de IntelTBB

II.6 Conclusion:

IntelTBB fournie aux développeurs un outil performant pour construire des applications desktop parallèle en exprimant le parallélisme en utilisant des méthodes de haut niveau, en outre IntelTBB fournie des méthodes de bat niveau pour plus de flexibilité dans les programmes qui ont le niveau de complexité assez élevé.

IntelTBB se développe encore pour répondre aux exigences de l'ingénierie du logiciel qui s'accroit de plus en plus. Pour connaitre les dernières nouvelles ainsi que pour télécharger les bibliothèques de IntelTBB visitez le site web: www. threadingbuildingblocks.org

III Une approche de refactoring pour le parallélisme : [4]

III.1 Introduction :

La façon la plus utilisée pour écrire un programme parallèle est de le faire d'une manière progressive, ensuite on passe à la révision du code tout en préservant le comportement ce celui-ci.

Les programmeurs préfèrent cette approche parce qu'elle maintient le travail, les versions déployable du programme à travers le processus du développement.

Cependant, l'approche de refactoring, est fastidieuse car elle effectue beaucoup de modifications dans le code, et demande aux programmeurs d'assurer les opérations parallèles.

III.2 Définition du refactoring :[10]

Le refactoring est une technique pour la restructuration d'un code existant, modifier la structure sans changer le comportement, le principe est d'effectuer des transformations successives sur le code existant, chaque transformation (appelée aussi un « refactoring ») modifie une petite partie du code et avec les successions des transformations on produit une restructuration significative, puisque les transformations sont petites elles sont donc moins exposées à l'erreur.

Le system garde les versions du programme pour chaque transformation afin de servir d'elles comme une documentation.

III.3 Les outils automatiques contre les outils interactifs :

Plusieurs outils ont été proposés pour réduire les efforts effectués par les programmeurs quand ils passent d'un programme séquentiel à un programme parallèle.

Deux types d'outils différents ont été proposés, les outils automatiques (qui transforment automatiquement un programme séquentiel en un programme parallèle) et les outils interactifs (les outils de refactoring). La différence fondamentale entre ces deux outils est le rôle du programmeur.

Quand un outil automatique travail, rend de grands résultats, malheureusement sans profiter des connaissances du programmeur, le compilateur limite la façon de l'exécution.

Aujourd'hui des compilateurs commerciaux ont réussi à faire du parallélisme dans des petites applications mais pas dans des grandes applications.

Et même si les compilateurs ont été amélioré, le compilateur doit encor coder la plupart du code lui-même, par contre, un outils interactive prend une approche différente comme un expert dans un domaine, le programmeur comprend les concept liées au parallélisme, ainsi le programme existant ne doit pas être changé, comme les données, les relations entre les parties du programme, l'algorithme utilisé et les structures de données, de cette façon, l'approche interactive combine les avantage des programmeurs(connaissances…) et des outils (vitesse, calcules…), le programmeur se charge de la partie création : sélection du code et les transformations, et l'outils fait le travail difficile en vérifiant les thread à travers les fonctions et les variables et la modification du programme.

Quand l'outil ne peut pas appliquer une transformation il fournit un feedback dans une interface visuelle da l'IDE comme éclipse ou visualstudio, et permet au programmeur de régler le problème.

Les outils de refactoring permettent aux programmeurs d'exploiter d'une façon continue une grande quantité de code avec la préservation du comportement initial.

III.4 Interactions entre l'outil de refactoring et le programmeur :

Un outil de refactoring à plusieurs point d'interaction avec le programmeur, il est responsable d'identifier les données partagées et les calcules intensifs et choisir un refactoring approprié.

Comme illustré dans la figure suivante (figure 3.12), le programmeur sélectionne d'abord du code séquentiel et choisit un refactoring, ensuite, l'outil analyse la sécurité de la transformation, par défaut l'outil de refactoring applique le changement si c'est une opération sécurisé (pas de changement dans le comportement du programme), si des pré-conditions ne sont pas rencontrées l'outil rend un signal, le programmeur peut décider d'arrêter le refactoring ou fixer le code malgré lesignal.

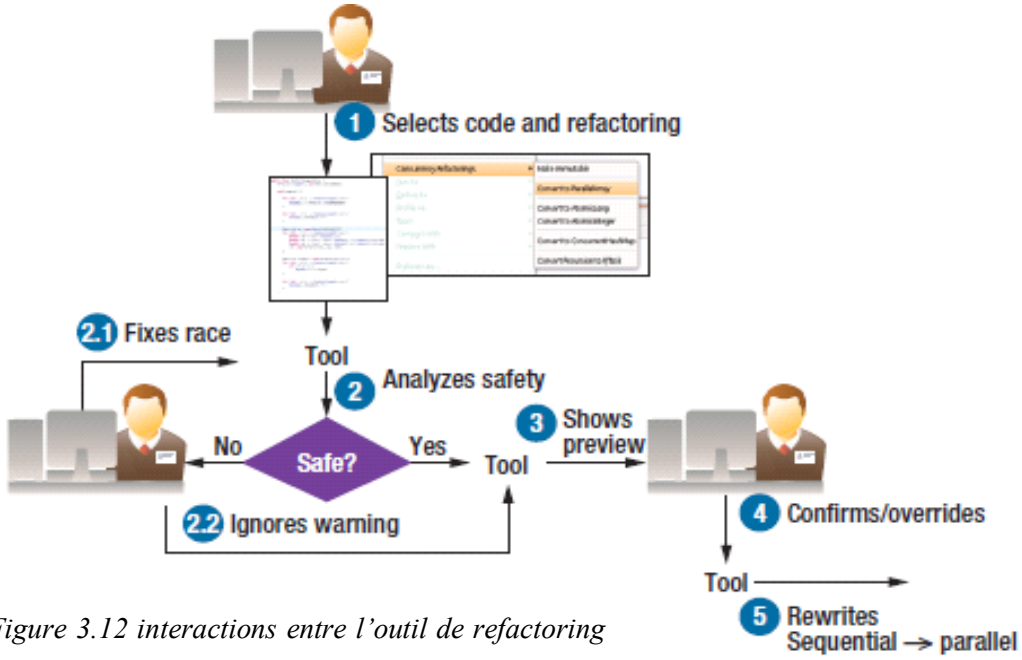


Figure 3.12 interactions entre l'outil de refactoring et l'utilisateur

Les IDE modernes comportent les outils interactives de refactoring, et ceci permet aux programmeurs de mener d'une façon efficace et l'optimisation de la performance du parallélisme si des fonctionnalités déjà existantes, en améliorant la productivité du programme, la portabilité du code et la performance.

III.4.1 Amélioration de la productivité du programme :

Après le refactoring d'un programme séquentiel, le programmeur prend généralement le nouveau code, mais pour plus de performance l'outil de refactoring prend et navigue dans les deux versions du code (code sequentiel et la transformation appliquée à ce code). Un IDE traite le refactoring comme un ensemble de transformations de classes, ordonne ces transformations pour servir comme une documentation et offre un programme compréhensible.

L'IDE fournit deux vue pour le même code, séquentiel et parallèle, le programmeur utilise la vue séquentiel pour comprendre le programme, fixer les bugs et ajouter d'autres fonctionnalités, et la vue parallèle pour la performance.

III.4.2 : Amélioration de la portabilité du code :

Le programmeur minimise le code pour plus de performance, cependant, ceci touche les caractéristiques du hardware et fait du code moins portable, et pour permettre au programme de s'exécuter dans une autre plateforme le programmeur doit effectuer des modifications sur ce

programme. L'IDE génère plusieurs implémentations pour la même transformation et le programmeur doit choisir la plus portable.

III.4.3 : Amélioration de la performance :

Quand l'outil de refactoring ne peut pas paralléliser un programme automatiquement, il fournie des informations que le programmeur peut utiliser pour paralléliser le programme lui même.

Le programmeur utilise ces connaissances en proposant des transformations et l'outil de refactoring propose plusieurs implémentations pour une même transformation et identifie les bottlenecks du programme après une analyse, le programmeur doit choisir une parmi ces transformations qui devrait être la plus performante, donc on peut dire que les outils de refactoring et les compilateurs sont complémentaires.

III.5 Les outils de refactiring pour le parallélisme :

Plusieurs outils de refactoring ont été proposés pour le parallélisme dans les applications desktop avec des bibliothèques paralleles et des framework :***Reentrance*** qui est un outil de refactoring récent de IBM, ***Robert Fuhrer*** avec le langage de programmation X10, ***un catalogue d'outils de refactoring*** proposé par Fortran enfin ***l'outil de refactoring de java*** que nous allons détailler par suite.

Les transformations parallèles de ces outils sont regroupées en trois catégories :

- Celles qui améliorent la sécurité de la transformation (thread safety).
- Celles qui améliorent la vitesse d'exécution.
- Celles qui améliorent la portabilité.

III.5.1 L'outil de refactoring de java:

Utilise les bibliothèques java et intégré dans éclipse, généralement utilise les trois catégories de transformations : la sécurité des transformations, vitesse d'exécution et la portabilité, ces transformations demande une analyse de contrôle du programme et les flux des données, ainsi les outils de refactoring doivent être capable de détecter les objets partagés dans les programme orienté objet.

Une démarche à suivre a été proposé par des chercheurs pour écrire un programme parallèle: écrire du code juste (tread safe), ensuite le rendre plus rapide dans l'exécution (multithreading), puis le rendre portable.

III.5.1.1 Le Refactoring pour thread safety:

Pour écrire un programme parallèle il faut définir les données variables à partager, soit synchroniser l'accès à ces données, les rendre non variables ou éliminer complètement le partage.

L'outil de refactoring de java support deux accès synchronisés : le premier convertie un**int** en un objet**AtomicInteger**, la classe**java.util.concurrent** fournie les opérations atomiques pour la mise à jour d'un champ, le deuxième convertie une table de hachage **HashMap** à un **ConccurentHashPap** qui est un thread safe par défaut pour travailler avec **HashMap.**

On peut rendre une classe entière thread safe en mettant toute ces attribut avec l'option ***final*** c'est à dire qu'ile ne peuvent pas être assignés en dehors des constructeurs. Une fois la classe est proprement construite (thread safe) elle peut être partagé entre les threads sans le besoin de synchronisation.

III.5.1.2 Le refactoring pour la vitesse d'exécution :

Lorsque le programme est « thread safe »le multithreading améliore la performance, on peut aussi gérer les threads manuellement ou automatiquement, cet outil propose deux solutions, la première est l'algorithme séquentiel divide-and-concer qui traite les sous problèmes parallèles en utilisant les frameworks***forkjointask*** (Une classe java qui gère un grand nombre de taches avec peu de thread), la deuxième utilise les tableaux java avec ***parallelArray*** (une classe java qui prend un pool de taches ou un tableau et fournit des opérations parallèles sur les éléments.)

Le refactoring change les données des tableaux et remplace les boucles par des éléments du tableau avec des opérations parallèles équivalentes en utilisant la bibliothèque ***ParallelArray*.**

III.5.1.3 : Le refactoring pour la portabilité :

Sacrifier la sécurité de la transformation (thread safety**)** pour la performance peut réduire la portabilité de l'application, particulièrement dans la synchronisation de bas niveau comme les verrous.

Une meilleure façon est d'utiliser les structures de données portables fournies par la bibliothèque parallèle.

Cet outil support deux solutions de refactoring pour l'amélioration de la portabilité, une convertie un**int** en un **atomicInteger** qui est une structure de donnée verrous-libre qui utilise les instructions hardware **compare-and-swap** (comparer et échanger) et l'autre convertie un champ **HashMap** à un **concurrentHashMap** qui est en lecture et écriture parallèle pour synchroniser l'accès à la table.

III.6 Conclusion :

Dans cet outil de refactoring on peut envisager ces point suivant : premièrement, l'outil de refactoring doit supporter les bibliothèques utilisées par les programmeurs, deuxièmement les IDEs et les bibliothèques sont essentiels pour les outils de refactoring, troisièmement un programme parallèle doit être portable et facile à maintenir, en fin l'outil de refactoring doit pouvoir se communiquer avec d'autres outils de parallélisme.

Il y a encore un besoin pour d'autres outils de refactoring pour améliorer la portabilité et la rentabilité, particulièrement dans l'industrie qui demande chaque jour plus de parallélisme dans les programmes.

IV Parallélisme avec Qt C++ [9]

IV.1 Introduction:

Les langages orientés objet sont les plus populaire et les plus utilisés par les développeurs d'application desketop aujourd'hui, si des nouveaux outils vont être développés pour amélioré la programmation des applications desketop ils doivent être introduits dans ces langages orientés objet.

Qt de C++ présente un concept puissant pour la programmation orienté objet. Les taches parallèles peuvent être implantées avec des mots clés sans changer la structure du programme, la définition et l'invocation des taches est comme la définition et l'invocation d'une fonction typique, les dépendances entre les taches sont spécifiées d'une façon simple et intuitive et ces dépendances sont gérées par le système.

IV.2 Présentation de Qt C++ :

Qt est un framewerk standard de C++ pour le développement d'applications de haute performance, en addition des bibliothèques Q++ existantes, Qt inclus des outils pour écrire des programme souple et améliorer la vitesse d'exécution. Les applications peuvent être exécutées dans des systèmes tels que XP, Mac OS et Linux. La comptabilité et l'internationalisation de la plateforme Qt à permis sa grande présence sur le marché.

IV.3 Le concept de tache :

Qt est utilisé pour les environnements desktop (KDE), et implémente le concept de tache. On peut produire le programme séquentiel équivalent en suppriment les mots clés liés au parallélisme ainsi 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.3.1 Définition et déclaration des taches :

Cette première tache prend une image et réduit ces dimensions, la déclaration d'une tache se fait avec le mot clé TASK :

TASK *void task1(String filename);*

La définition reste comme une fonction C++:

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

Les variables dans la fonction TASK sont privées et les variables de classe sont partagées.

IV.3.2 : Invocation d'une tache :

L'invocation d'une tache est comme l'invocation d'une fonction typique C++, avec deux différences :

Premièrement, la tache s'exécute d'une façon asynchrone avec le programme qui l'appelle, deuxièmement, l'invocation retourne un objet TASKID.

Ce code suivant invoque une tache pour chaque image :

```
List images = ...;

for (int i = 0; i < images.size(); i++)
```

*TaskID *id = task1(images.at(i));*

IV.4 Dépendances entre les taches:

Les dépendances se fassent avec le mot clés (dependson), par exemple la tache 4 doit attendre les taches 2 et 3, et les taches 2 et 3 doivent attendre la tache 1 :

```
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))
dependsOn(id2, id3);
        }
```

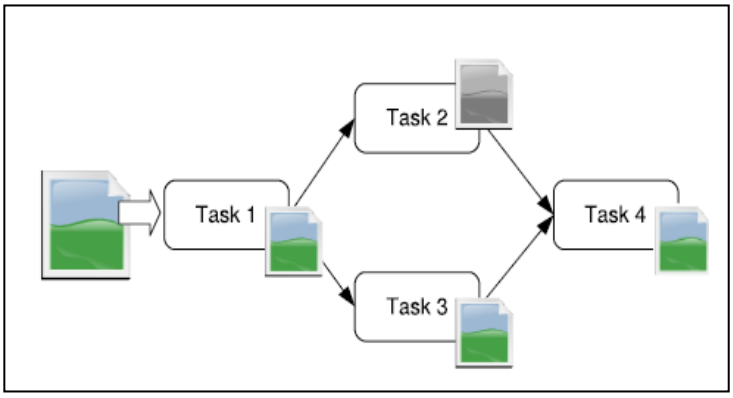


figure 3.13 dépendances des taches

On remarque que le programme n'a pas besoin des mécanismes de synchronisation manuelle pour gérer les dépendances.

On note aussi qu'on peut gérer plusieurs TASKID collectivement avec le mot clés TASKIDgroup.

IV.5 : Résultats et valeurs de retour :

Quand ont invoque une classe d'une manière asynchrone il est important de savoir quand elle se termine, la tache retourne un résultat ou pas, elle est invoquée dans les deux cas.

*TaskID *idReturn = taskWithReturn();*

*TaskID *idNoReturn = taskWithNoReturn();*

IV.5.1 Approche avec blocage :

pour les taches qui retourne une valeur le résultat est accessible par la méthoderesult(), si la tache est accomplit result() retourne immédiatement le résultat, et le programme appelant se bloque jusqu'à la fin de la tache :

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

IV.5.2 Approche sans blocage :

L'approche avec blocage pose un problème pour les programme interactives qui se basent sur les événements, pour régler ce problème, le concept de tache fournie une approche sans blocage, quand la tache est invoquée une liste de slots est spécifiée en utilisant la clause **notify**, ces slots sont invoqué automatiquement quand la tache soit complétée.

*TaskID *idReturn = taskWithReturn()*

notify*(mySlot(TaskID*));*

IV.6 : Les différents types de tache :

On peut envisager plusieurs types de tache, quelques taches s'exécutent rapidement, quand à autres demandent plus de temps, quelques taches s'exécutent une seul fois et d'autres s'exécutent plusieurs fois par des threads différents, tous ces taches sont utilisées dans un seul modèle, et ont un seul globalID accessible en utilisant TASK ::globalID().

IV.6.1 Taches one-off :

Quand une tache one-off est invoquée, elle sera empilée dans une file et exécutée par les threads worker, la sémantique de cette tache est qu'elle s'exécute du début jusqu'a la fin dans un thread worker, quand la tache s'achève une autre tache se suit pour le même thread worker.

IV.6.2 Multitâches :

Dans le parallélisme de données, on utilise le même calcule plusieurs fois sur des données différentes. On préfère invoquer une tache une seul fois et cette tache sera invoquée plusieurs fois automatiquement.

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

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

multi-task size représente le nombre des sous taches dans une multitâche.

relative ID représente l'identifiant de la sous tache avec le résultat de la sous tache 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.6.3 : les taches interactives :

Ce sontdes taches avec une longue durée d'exécution et qui se bloquent fréquemment. Elles se déclarent comme les taches one-off mais sont invoquées avec le mot clés *interactive*

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

Les taches interactives s'exécutent dans un thread interactive séparé, elles s'exécutent aussi sans l'attente des threads worker libres, et fournissent une repense rapide pour l'activité interactives importantes.

IV.7 Implémentation :

I.7.1 Planification des taches :

La *figure 3.13* montre comment le système gère un tache àprés sont invocation. Quand la tache est invoquée elle sera empilée dans une file (etape1), la tache sera ensuite placée dans un pool de tache (etape2), un nouveau TaskID est retourné(etape3), ceci permettra au programme appelant de continuer son exécution (etape4), et la tache s'exécute d'une façon asynchrone dans un thread worker(étape5).

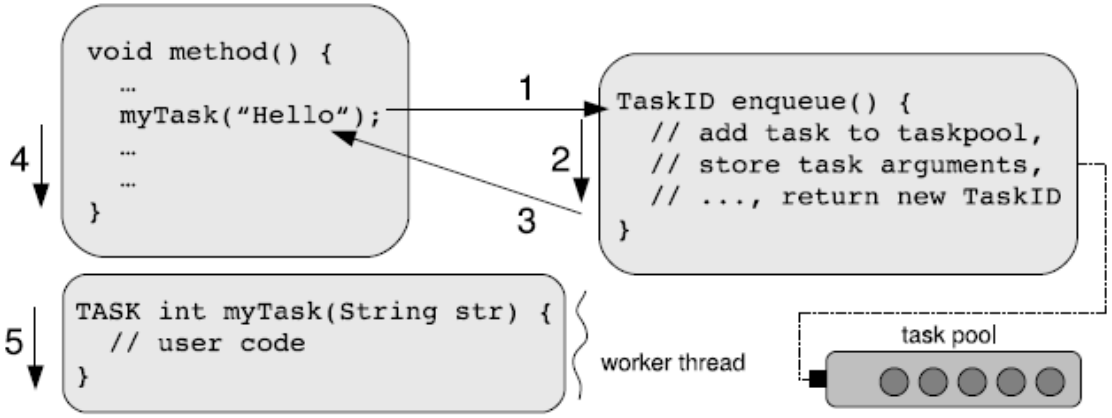


Figure 3.14 planification des taches par Qt C++

IV.7.2 Système d'exécution:

La figure 3.14 montre le système d'exécution de l'outil Qt de C++.

Empiler la tache :

Quand une tache est empilée, des liens seront établis aux slots enregistrés (**notify**), si la tache dépend d'autres taches, elle sera enregistrée avec un les taches en attentes (waiting tasks). Une tache

non interactive prête peut être exécutée par n'importe quel thread worker libre, et une tache interactive s'exécute dans un nouveau thread worker interactive.

Thread Worker est prêt :

Toutes les taches à l'exception des taches interactives serons exécutées par un thread worker, au début le thread worker exécute toutes les taches de sa propre liste prête privée, ensuite passe à la liste partagée, si la tache est réservée pour un autre thread worker, elle sera empilée dans la liste privée de ce thread worker, ensuite le thread worker se bloque s'il n y a aucune tache dans la pile privée ou partagée.

Accomplissement des taches :

Quand une tache termine son exécution, le thread manager de pool de tache est responsable de la mise à jour de la liste des taches en attente, la tache sera soit empilée dans une liste prête partagée soit sera exécutée par un nouveau thread worker interactif.

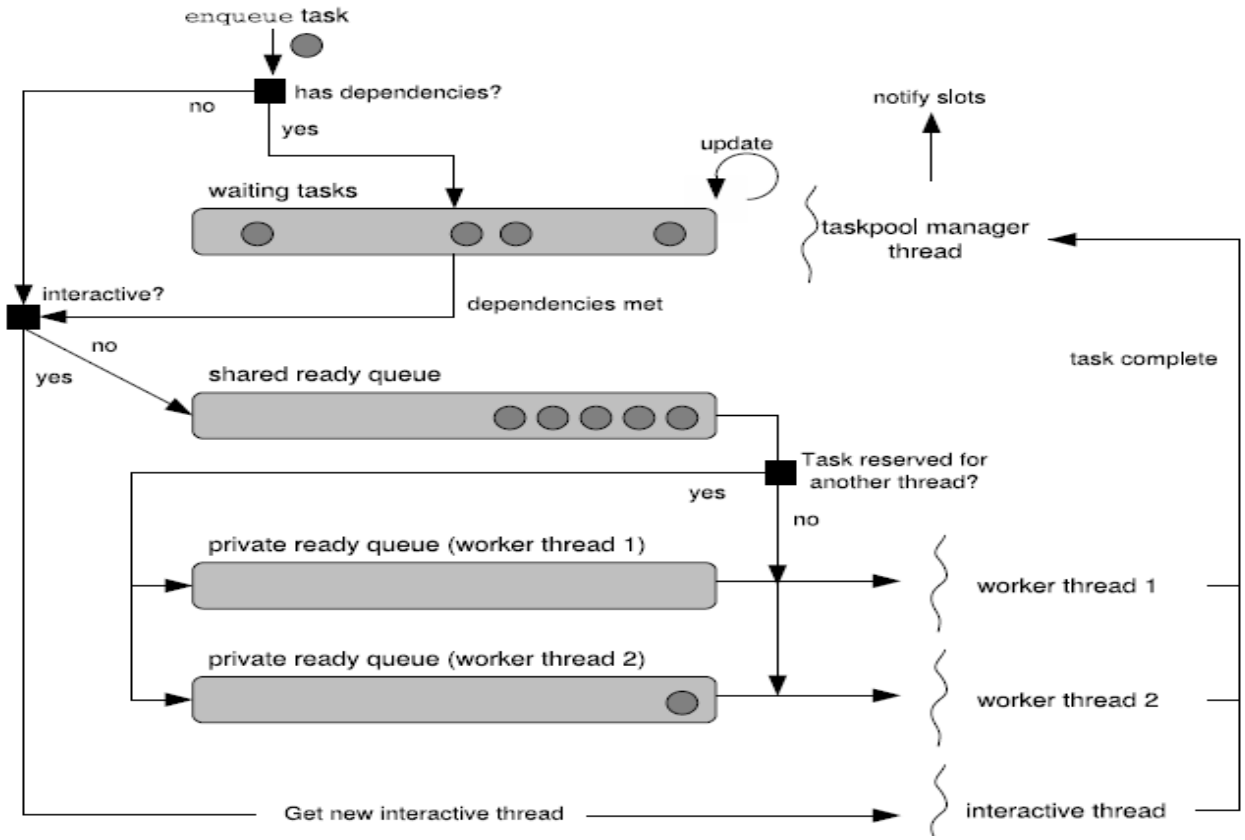


Figure 3.15 Système d'exécution de C++

IV.8 Conclusion :

Cet outil propose un concept de tache pour les programmes orienté objet dans les systèmes à mémoire partagé, le programmeur n'a pas besoin de modifier le programme séquentiel pour avoir le programme parallèle, des mécanismes de synchronisation intuitives aident dans la mise en œuvre des mécanismes de synchronisation délicats en particuliers les mécanismes de synchronisation sans blocage pour garder l'interactivité des programme qui se basent sur les événements.

V. parallelTask [8]

V.1. Introduction :

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

V.2 Définition et invocation des taches :

Paratask encapsule les taches au niveau des méthodes, la définition se fait avec le mot clés TASK, voici une définition d'une tache qui détecte les contours d'une image.

```
TASK public Image edgeDetectTask(Image i)

{ // detect the edges }
```

Cet événement handleur montre l'invocation de cette tache :

```
public void actionPerformed(ActionEvent e) {

... for (Image image: selectedImages) {

TaskID<Image> result = edgeDetectTask(image); ... } }
```

Invoquer un tache parallèle avec ParaTask ce fait de la même manière d'invoquer une méthode standard de java, à l'exception qu'elle s'exécute en parallèle avec le programme appelant.

TaskID représente la valeur de la tache invoquée.

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

Dans une implémentation séquentiel le thread GUI détecte les contours entiers de l'images l'application ce bloque en ce moment. Cependant dans le mode parallèle le thread GUI place les taches dans une file et retourne à la boucle de l'événement (event loop). ensuite Les taches vont s'exécutées par un groupe de threads.

Paratask crée et gère automatiquement un nombre idéal de threads pour garder chaque unité de calcule occupé, donc invoquer plusieurs taches est plus efficace que créer un thread pour chaque traitement.

Maintenant que la tache est chargée pour un autre thread, le thread GUI doit savoir le résultat d'exécution de la tache.

Image i = result.getResult();

Si la tache est déjà complétée, il n y aura aucun problème, le résultat est disponible sinon, le thread qui invoque cette tache se bloque jusqu'à ce que la tache 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 tache :

Voici une solution qui propose une approche sans blocage :

```
publi 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) et après se rend compte de l'accomplissement de la tache pour mettre à jour la l'application en utilisant la clause **notify.**

V.3 Les dépendances :

Les taches peuvent se dépendent entre elles, par exemple appliquer plusieurs filtres successifs pour une images, un nouveau filtre doit être appliqué après que l'ancien soit terminé

Avec paratask on déclare les dépendances avec le mot clés ***dependson***.

```
1    for (Image image: selectedImages) {
2    TaskIDGroup history = historyMap.get(image);
4    TaskID result = blurTask(image)
5    notify(updateGUI(TaskID))
6    dependsOn(history);
7
8    history.add(result);
9    }
```

Chaque image a un historique de filtres(2), chaque fois qu'on applique un nouveau filtre à une image une nouvelle tache sera créée ce filtre sera appliqué lorsque les filtres précédents sont déjà appliqués (ligne6) et sans cette dépendance le filtre sera appliqué à l'image originale.

Lorsque l'image est invoquée elle sera ajoutée à l'historique de l'image (ligne8) pour que les futures taches attendent son accomplissement.

V.4 Les taches interactives :

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

Paratask définie des taches interactives comme suis :

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

```
}
```

Paratask traite les taches interactives comme les taches standards, la différence c'est que la tache interactive ne sera pas empilée dans la file du thread worker.

V.5 Les multitâches :

Les multitâches en ParaTask permettent de déterminer la position d'une sous tache dans le groupe (ligne 2 et 4) et une barrière de synchronisation entre les sous taches (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   }
```

V.6 Conclusion:

Le but de ParaTask est de construire de vrais applications parallèles orientés objet, minimiser la restructuration du code séquentiel, maximiser la réutilisabilité du code, introduire les concepts du parallélisme sans affecter la performance. On peut trouver son utilisation dans des domaines différents tels que applications graphique, l'imagerie, interaction avec le web …

Conclusion

Nous avons vue au cours de ce chapitre comment l'ancienne approche de parallélisme qui utilise les threads n'a pas pue répondre aux besoins de l'ingénierie du

logiciel qui demande de plus en plus des calcules intensifs et des quantités gigantesque de données, ainsi que l'arrivé des machines muticore à influencer l'ingénierie du logiciel.

Nous avons vue comment les applications desketop se diffèrent des autres applications de manière qu'elles s'exécutent sur des machines et des plateformes différentes, et plus précisément les applications orientés objets qui sont considérées comme les plus populaires et les plus utilisés par les développeurs aujourd'hui.

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

Chapitre 4
ParaTask de JAVA

Introduction

Dans le chapitre précédent nous avons vu un ensemble d'outils qui permets la construction des programmes parallèles à travers des programmes séquentiels déjà existants, parmi ces outils nous avons choisis la solution proposée par java ParaTask, et comme nous avons déjà dis auparavant cette solution permet de paralléliser des applications GUI orientés objet.

Le but de ce chapitre est de mieux expliquer la programmation parallèle avec l'outil ParaTask, pour cela on commence par un exemple introductif qui montre la façon dont les tâches parallèles peuvent être créées, ensuite un deuxième exemple qui montre la différence entre les types de tâches utilisées par ParaTast, enfin un exemple pour le traitement d'image (détection de contours), les codes sources de ce chapitre seront disponible dans l'annexe réservé aux codes sources.

I. Exemple 1: invocation des tâches:

dans cet exemple nous allons expliquer comment procéder pour invoquer une tâche one-off, multitâche et tâche interactive qui peuvent s'exécuter en parallèle, le programme consiste à

afficher une chaine de caractère à travers la méthode hello().

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

Les trois méthodes suivantes représentent respectivement la définition de la 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");
   }
```

La première chose à remarquer est la réutilisabilité du code, les trois méthodes réutilisent le code séquentiel en appelant la méthode hello(), la deuxième chose à remarquer est 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 sequentiel, les trois autre méthodes qui sont déclarés 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.*

La chose la plus importante à remarquer est que 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.

II. Exemple 2 : différence entre les types de tâches

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

La méthode qui effectue cetraitement 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.

II.2. 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 du 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 du chaque traitement, ce signifie que l'application est bloqué pendant 16 seconds.

II.3 Tâches one-off:

La première étape consiste à définir la tâche, et pour la réutilisabilité du code on vas encapsuler dedans la méthode séquentiel 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: 6.994 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 6.9 seconds.

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.

II.4 Multitâches:

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

La définition de la multitâche est données 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 le groupe. Maintenant on passe 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 le même de celui de la tâche One-off, ceci est du en faite à l'utilisation d'un nombre fixe thread worker, cependant les multitâche simplifie l'écriture des codes parallèles puisque TaskIDGroup qui contient les TaskIDs est retourné automatiquement.

II.5 Tâches interactives:

Quoique les tâches One-off sont les plus utilisés dans ParaTask, ne présentent pas la bonne solution pour cet exemple, puisque la méthodes webAccess() ne fais pas un calcule compliqué et le thread worker passe son temps dans l'attente du réseau. Cet ici que les tâches interactives sont les plus appropriées: les calcules 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 calcule n'est pas compliqué on préfère

exécuter toutes les tâches d'une façon concurrente, puisque les threads worker se bloquent en attendant les 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 lent

III. Exemple 3 : détection des contours d'image

Dans cet exemple on va essayer de traiter une image afin de détecter les contours. On note que la détection des contours d'images est un sujet vaste dans le domaine de traitement d'images qui représente l'un des modes les plus importants dans la communication à l'ère actuelle. On trouve aussi ses applications dans des domaines très variés tels que les télécommunications, la médecine, biologie, astronomie, la météorologie, l'architecture …etc.

Le but de cet exemple n'est pas de présenter les détails techniques de détection des contours d'image, pour plus de détails, de nombreux ouvrages ainsi que des sites web sont disponibles sur internet sur ce sujet.

On trouve plusieurs méthodes pour la détection des contours d'image parmi ces méthodes, la méthode de Canny qui est décrite par l'algorithme de détection suivant :

III.1 Algorithme séquentiel de Canny pour la detection de contours d'images:

```
Fonction Canny (image, imagerésultat, w, h)
Debut
    Pour i ←1 jusqu'à i < h faire
      Debut
        Pour j←1 jusqu'à j < w faire
```

```
Debut
        TAB [i] [j] ← image ( i , j)
Fin
  Fin

Pour  i  ← 1 jusqu'à  i <h/2 faire
  Debut
    Pour  j ← 1  jusqu'à  j <w/2  faire
Debut
          X  ← TAB [i] [j] + TAB [i] [j+2]
          Y ← TAB [i] [j] - TAB [i+2] [j]
G   | X | + | Y |
                Imagerésultat [i] [j]  ←  G
       Fin
  Fin
Fin
```

Où : *w* est le nombre de colonnes et *h*est le nombre de lignes de l'image.

Le programme séquentiel java correspondant à cet algorithme est donné dans l'annexe réservée aux codes sources.

L'exécution du programme séquentiel sur une machine à deux processeurs, a donné le résultat suivant :

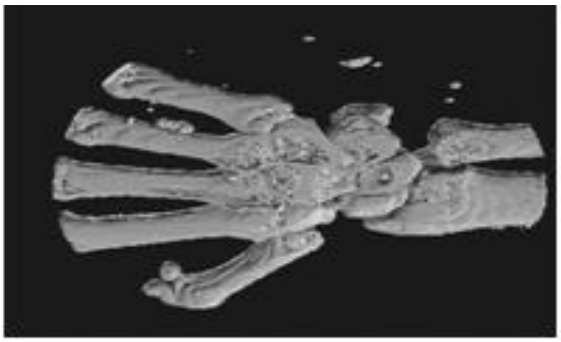


Figure4.1 : image source1 : 1 mains

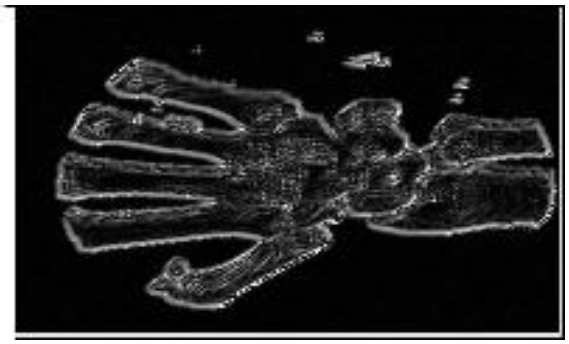
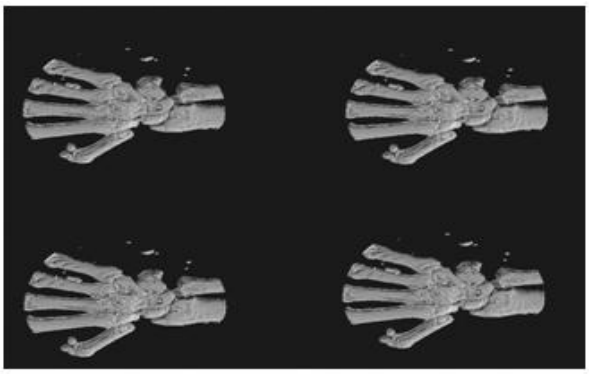


Figure4.2 : Solution séquentielle de détection de contours image1.



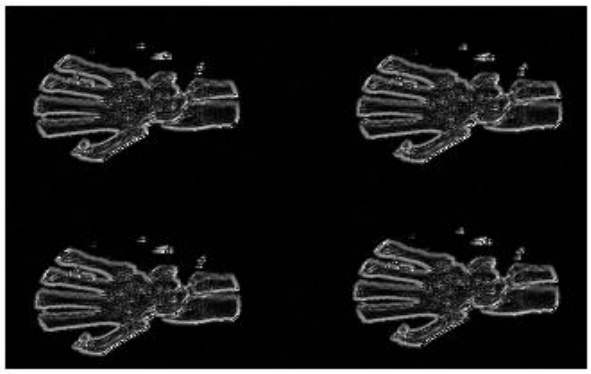
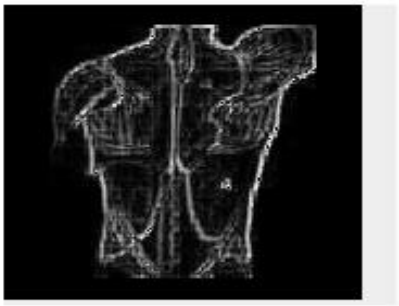


Figure4.4 : Solution séquentielle de détection de contours image2.

Figure4.3 : image source2 : 4 mains



T *Figure4.5 : image source 3 : dos* .0⁴ *Figure4.6 : Solution séquentielle de détection de contours image3.*

Temps d'exécution séquentielle de l'image 2 : 0.08[illegible] secondes

Temps d'exécution séquentielle de l'image 3 : 0.027 secondes.

III.2 Solution parallèle :

On regardant de plus près l'algorithme de détection de contour, on constate qu'il n'y a aucune dépendancesur le traitement des données ni le besoin des communications de résultats intermédiaires. Nous allons utiliser la décomposition par domaine. L'image sera partagée en plusieurs portions.Chaque portion sera affectée à une tâche qui calcule les contours de la portion d'image, à la fin du traitement on récupère les résultats de chaque tâche et on affiche l'image finale. Pour cela nous allons utiliser une répartition des données par bloc et une répartition cyclique.

III.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 la tâche.

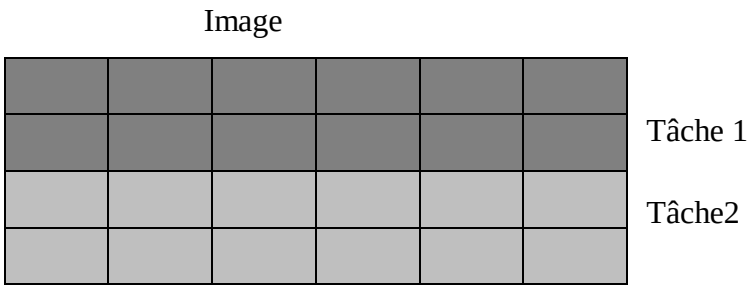


Figure4.7 répartition des données par blocs.

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*** *BlockParallel(BufferedImage img,BufferedImage imgr,* ***int*** *debut,* ***int*** *fin)*

```
{
long start = System.currentTimeMillis(); //initialiser le temps
d'execution
Canny (img, imgr, debut , fin);

long time = System.currentTimeMillis() - start; // calculer le temps
d'execution
long id = Thread.currentThread().getId();     // recuperer le thread
en cours
System.out.println(time / 1000.0 + " seconds, pour thread["+id+"]");
}
```

Dans la fonction Canny ,**img** représente l'image qu'on veut traiter et **imgr** est l'image résultat (détection des contours).

Après avoir défini la tâche à exécuter en parallèle, il ne reste plus qu'à invoquer cette dernière. Le nombre de tâche 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.

Dans notre cas deux tâches seront invoquées:

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

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

L'exécution de ce programme à donné les résultats suivant :

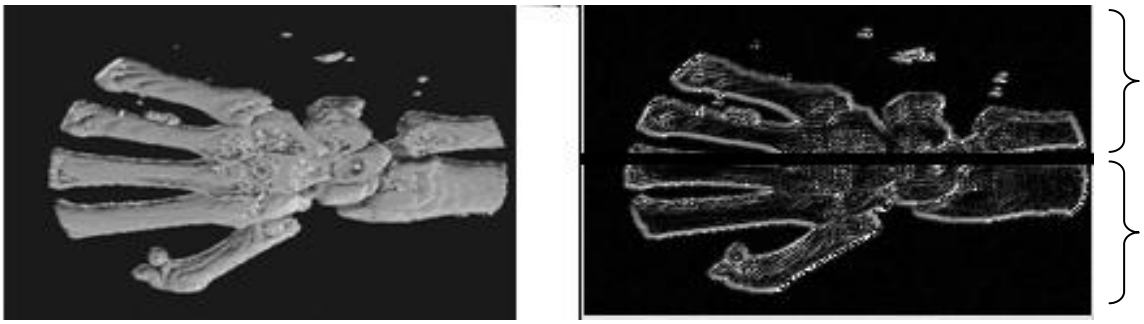


Figure 4.8: Contours d'image1 avec traitement parallèle sur 2 blocs de données Avec chevauchements

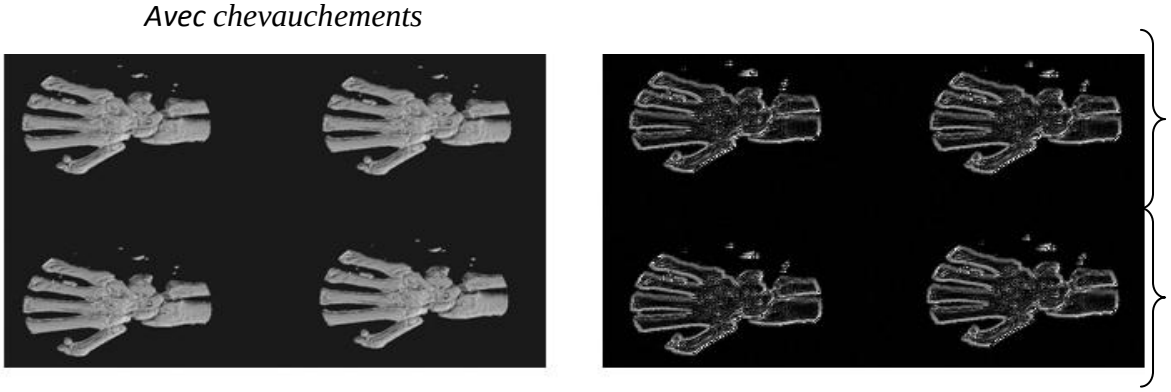


Figure 4.9: Contours d'image2 avec traitement parallèle sur 2 blocs de données Avec chevauchements

Figure 4.10: Contours d'image3 avec traitement parallèle sur 2 blocs de données

Les contours sont détectés, mais il y a un problème sur les bordures des blocs qui se manifeste par un vide entre chaque bloc.

Si on revient à l'algorithme séquentiel donné précédemment :

Pour i ⟵ 1 jusqu'à i < h/2 faire
Debut
Pour j ⟵ 1 jusqu'à j < w/2 faire
Debut

X ← *TAB [i] [j] + TAB [i] [j+2]*

Y ← *TAB [i] [j] - TAB [i+2] [j]*

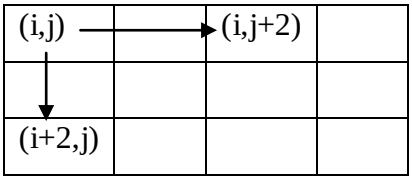
G ← *| X | + | Y |*

Imagerésultat [i] [j] ← *G*

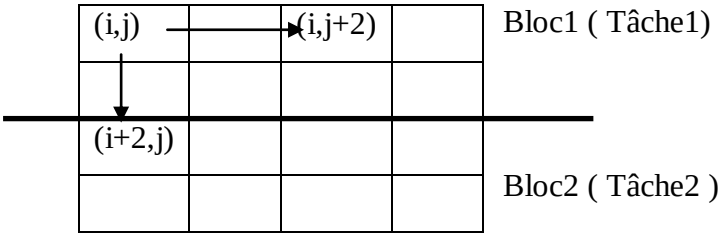
Fin

Fin

On remarque que le pixel (i,j) à un rapport avec le pixels (i+2, j), et le pixel (i, j+2) .



Lorsqu'on divise l'image en deux blocs comme le montre le schéma suivant :



Les deux dernières lignes du bloc 1 affectées à la tâche 1 ne peuvent pas être calculées car les données à utiliserse situent dans le bloc2 .

La solution consiste à ajouter au bloc1 les deux premières lignes du bloc2 afin de permettre à la tâche 1 d'effectuer correctement son traitement de détection de contour. Ceci va créer un léger chevauchement dans la répartition des données et l'invocation des tâches deviendra comme suit :

// on ajoute 2 pour la fin de la portion 1, h/4+2.

```
TaskIDGroup g=newTaskIDGroup(4);

        TaskID id1 = taskgris(img,imgr,0,h/2+2);
              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.

Processeur 1 **processeur2**

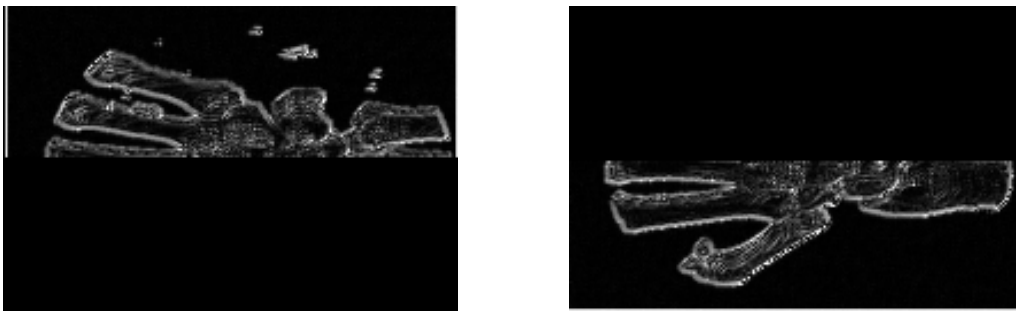


Figure 4.11 : image1,*Répartition de données par bloc*, résultats de chaque processeur

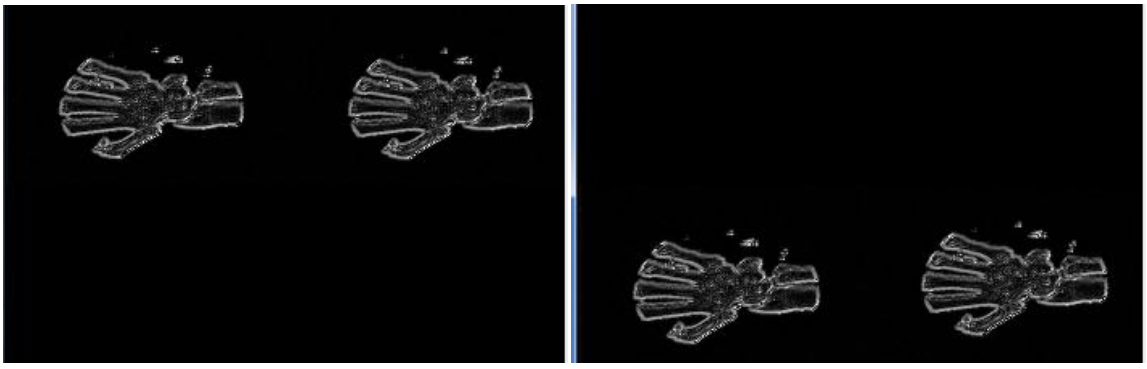


Figure 4.12 : image2, *Répartition de données par bloc*, résultats de chaque processeur

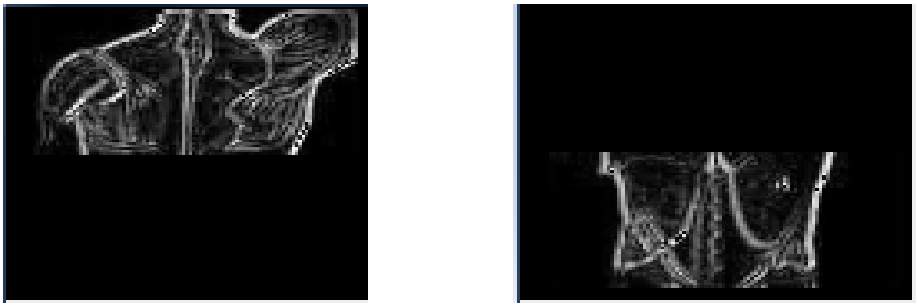


Figure 4.13 : image3, *Répartition de données par bloc*, résultats de chaque processeur

Maintenant on passe aux résultats retournés par les deux processeurs, on remarque que les contours sont bien détectés, plus de chevauchement et un temps de réponse moins élevée.

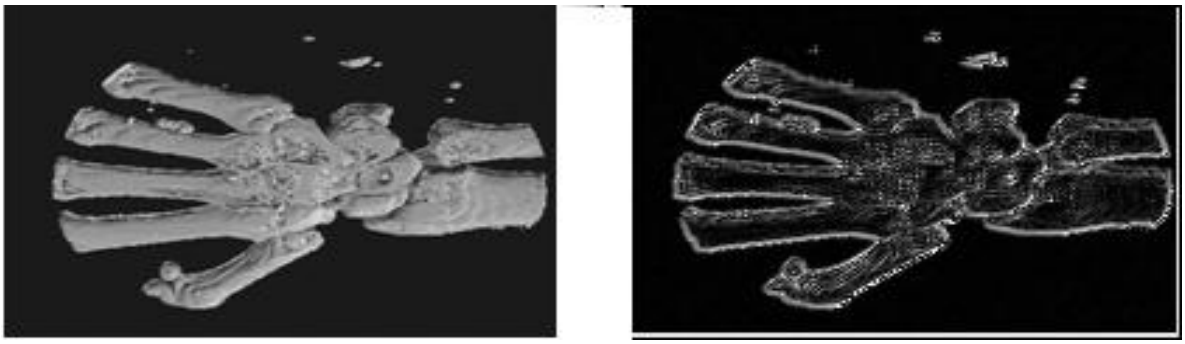


Figure 4.14 : Détection de contours image 1.
Solution parallèle avec répartition de données sur 4 blocs plus chevauchement debordure.

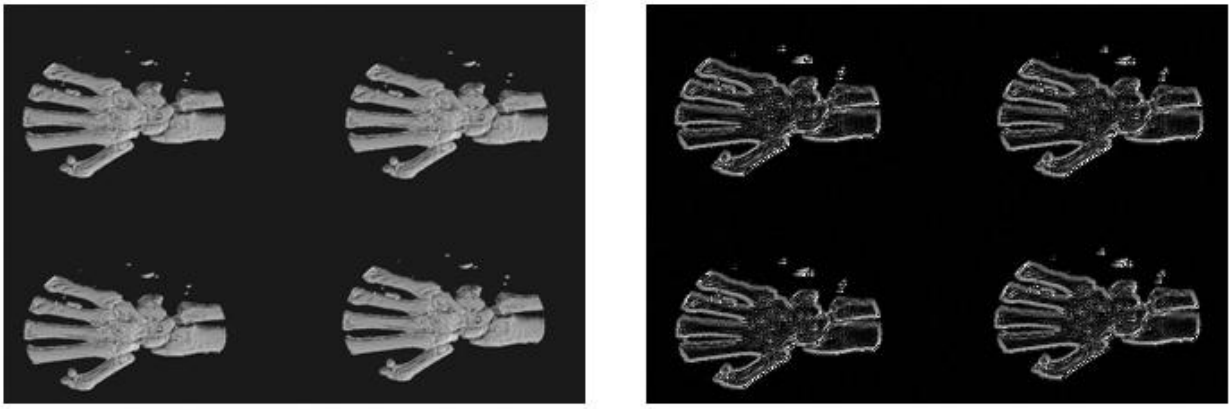


Figure 4.15 : Détection de contours image 1.
Solution parallèle avec répartition de données sur 4 blocs plus chevauchement de bordure.

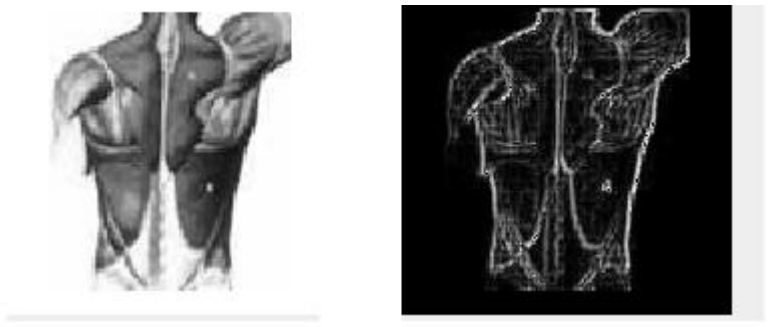


Figure 4.16 : Détection de contours image 2.
Solution parallèle avec répartition de données sur 4 blocs plus chevauchement de bordure.

Les résultats des traitements sont donnés comme suit :

Image1 : Temps d'exécution pour la détection de contour

```
0.032 seconde, pour thread[28]
0.027 seconde, pour thread[29]
```

On constate que les traitements se sont effectués sur deux threads car notre système possède deux processeurs.

La durée d'exécution totale est le maximum des temps d'exécution des threads ; dans notre cas c'est le thread 28 avec la valeur de 0.032seconde.

Image2 : Temps d'exécution pour la détection de contour

```
0.063 seconde, pour thread[26]
0.063 seconde, pour thread[27]
```

La duré d'exécution totale de l'image 2 est max (0.063,0.063)= 0.063 seconde.

Image3 : Temps d'exécution pour la détection de contour

```
0.015 seconde, pour thread[26]
0.017 seconde, pour thread[27]
```

La duré d'exécution totale de l'image 3 est max (0.015, 0.017)= 0.017 seconde.

III.2.2 Répartition cyclique des données.

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

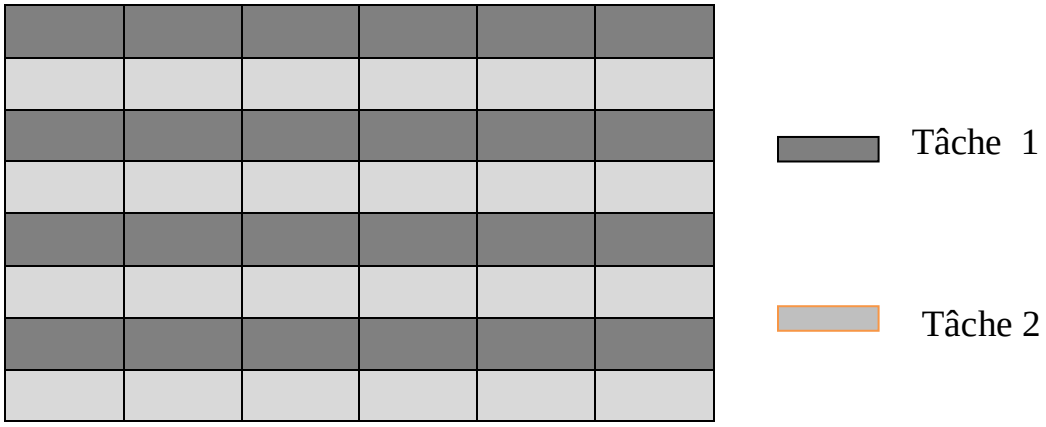


Figure 4.17 : Répartition cyclique des données de l'image

définir les entiers début 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.

```
TASKpublicstaticvoidCYCLICedge(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.Canny(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 qu'à invoquer cette dernière :

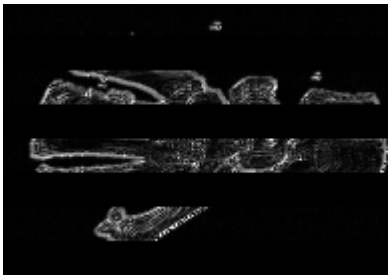
```
TaskIDGroup g=newTaskIDGroup(4);
                        TaskID id1 = CYCLICedge (img,imgr,h,0);
                        g.add(id1);

                        TaskID id2 = CYCLICedge (img,imgr,h,1);
                        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.

Processeur 1 **processeur2**



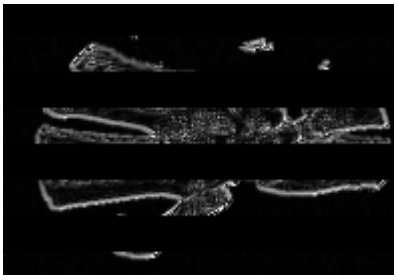


Figure 4.18 : image1,répartition cyclique des données, résultats de chaque processeur

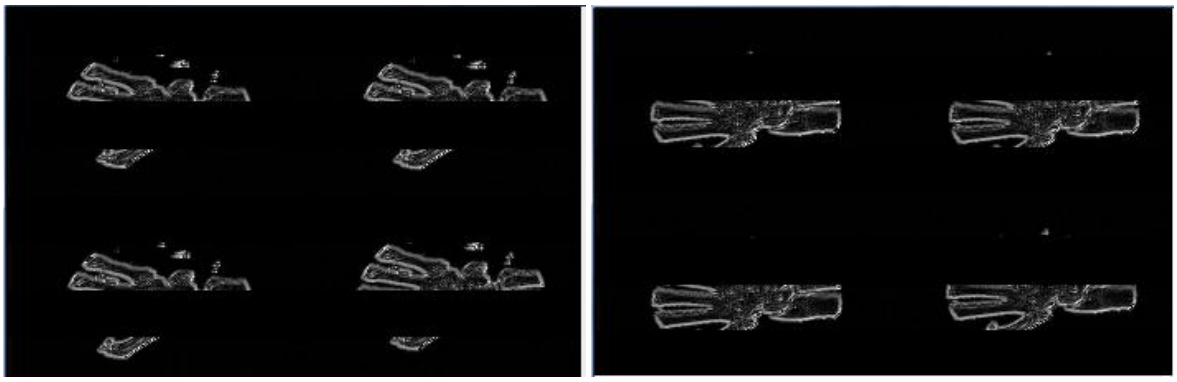


Figure 4.19 : image2,répartition cyclique des données, résultats de chaque processeur

Figure 4.20 : image3,répartition cyclique des données, résultats de chaque processeur

La détection des contours entiers sont détectés comme nous avons vue précédemment.

Image1 :

```
0.008 seconds, pour thread[30] cycle 0
0.012 seconds, pour thread[29]cycle 0
0.007 seconds, pour thread[30]cycle 1
0.007 seconds, pour thread[29]cycle 1
0.002 seconds, pour thread [30]cycle 2
0.004 seconds, pour thread [29]cycle 2
0.016 seconds, pour thread[30]cycle 3
0.016 seconds, pour thread[29]cycle 3
```

Temps d'[illegible]=0.033.

Temps d'[illegible]=0.039.

Le temps d'exécution total de l'image 1 est max (0.033, 0.039)= 0.039.

Image2

```
0.016 seconds, pour thread[25]cycle0
0.016 seconds, pour thread[26]cycle0
0.016 seconds, pour thread[25]cycle1
0.016 seconds, pour thread[26]cycle1
0.015 seconds, pour thread[25]cycle2
0.015 seconds, pour thread[26]cycle2
0.016 seconds, pour thread[25]cycle3
0.016 seconds, pour thread[26]cycle3
```

Temps d'exécution du thread 25 = 0.016+0.016+0.015+0.016=0.063

Temps d'exécution du thread 26 = 0.016+0.016+0.015+0.016=0.063

Le temps d'exécution total de l'image 2 est 0.063

Image3 :

```
0.005 seconds, pour thread[28]cycle0
0.005 seconds, pour thread[27]cycle0
0.002 seconds, pour thread[28]cycle1
0.002 seconds, pour thread[27]cycle1
0.005 seconds, pour thread[28]cycle2
0.005 seconds, pour thread[27]cycle2
0.005 seconds, pour thread[28]cycle3
0.003 seconds, pour thread[27]cycle3
```

Temps d'exécution du thread 28 = 0.005+0.002+0.005+0.005=0.017.

Temps d'exécution du thread 29 = 0.005+0.002+0.005+0.003=0.015.

Le temps d'exécution total de l'image 3 est max (0.017, 0.015)= 0.017.

III.2.3. Calcul de l'accélération et de l'efficacité :

Pour récapituler les différents traitements que nous avons réaliser 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.

	Image1 : 1main	Image 2 :2mains	Image3 : dos
Tems d'execution sequentiel	0.043	0.082	0.027

Tems d'execution Parallèle (block)	0.032	0.063	0.017
Accélération(block)	1.34	1.30	1.58
Efficacité(block)	0.67	0.65	0.79
Tems d'execution Parallèle(cyclique)	0.039	0.063	0.017
Accélération(cyclique)	1.10	1.30	1.58
Efficacité(cyclique)	0.55	0.65	0.79

Figure 4.21 tableau récapitulatif des temps d'exécutions de chaque solution.

Conclusion :

Nous avons vu dans ce chapitre quelques détails pratiques pour construire des programmes parallèles avec l'outil ParaTask de java. Nous avons également montré comment déclarer et invoquer les tâches parallèles, quoique les exemples que nous avons présenté dans ce chapitre ne sont pas assez compliqués qui nécessitent un traitement parallèle. ParaTask sera très utile dans la construction des applications qui demandent beaucoup de calcul, les applications interactives qui nécessitent beaucoup de synchronisation et de communication entre les tâches ainsi que les communications avec l'extérieur.

Enfin, vue sa facilité d'utilisation, ParaTask constitue une bonne solution pratique pour construire des applications GUI parallèles.

Annexe A : environnement de développement

I. Le langage de programmation Java :

Un langage de programmation orienté objet mis au point par Sun Microsystems. Sa caractéristique principale est qu'il est indépendant de toute plateforme, il est possible

d’exécuter des programme Java sur tous les environnements procédant une machine virtuel java (JVM), ce concept est à la base du slogan de Sun pour java : WORA (Write Once Run Anywhere). Sun fournit aussi gratuitement un ensemble d’outils et d’API pour permettre le développement de programmes avec ce langage, ce Kit est nommé JDK (Java Developement Kit).

II. IDE Eclipse :

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

III. Parallel Task : un Plug-in de Eclipse

Le plug-in ParaTask a été développé pour Eclipse version 3.5.1 ou plus, vous pouvez le télécharger de l'adresse suivant:

http://www.ece.auckland.ac.nz/~parallel/ParallelIT/updatesite/

III.1 Installation du plug-in:

1. ouvrez eclipse
2. cliquez sur help >>> install New Software
3. cliquez sur add
4. copiez et collez l'adresse URL du plug-in ParaTask dans le champ texte
5. cliquez sur ok
6. sélectionnez PTJava Text Editor ensuite cliquez sur Next pour installer le plug-in

III.2 créer un projet ParaTask:

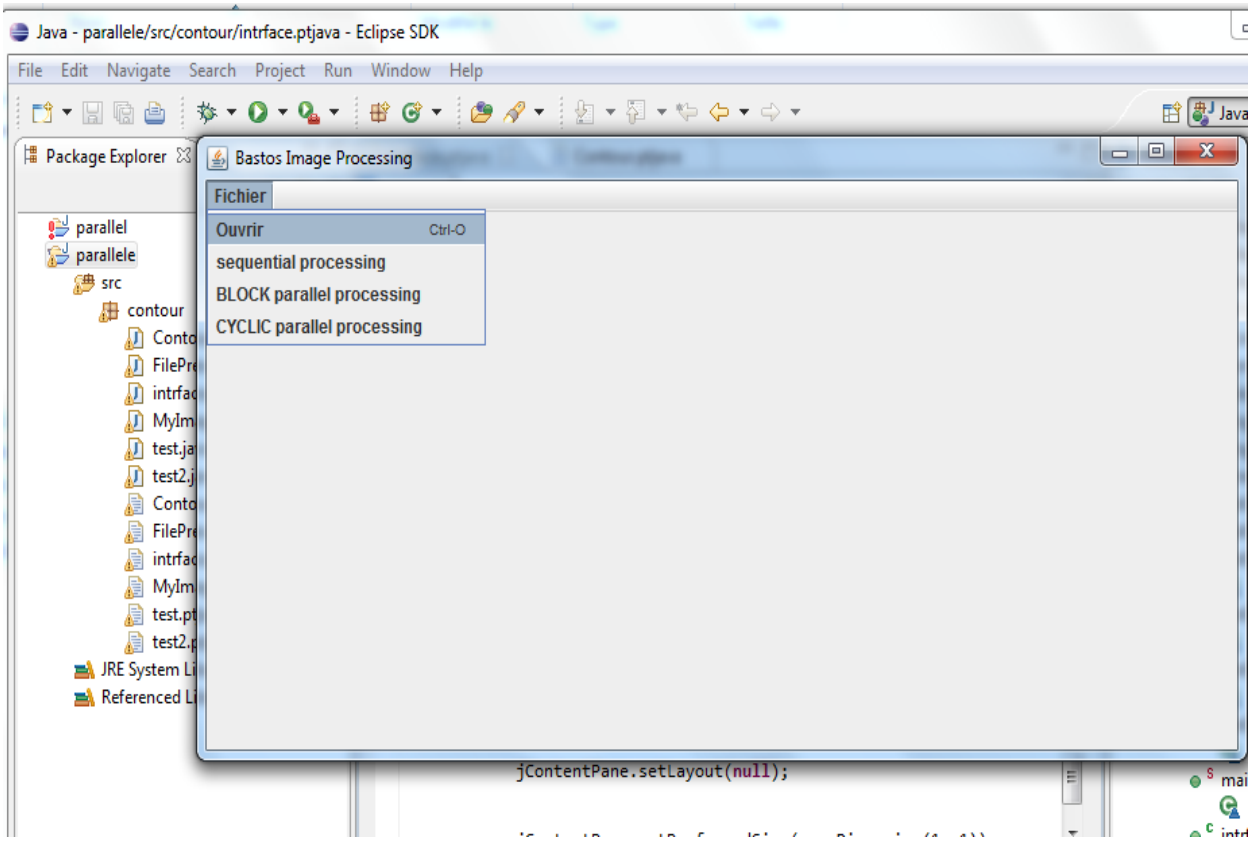
Après avoir installé ParaTask redémarrez éclipse pour qu'il prend en considération le nouveau plug-in installé, ensuite comme tout projet java ordinaire cliquez sur File >>> New

Project, et au lieu de choisir "Java Project" choisissez" PTjava Project", la même chose pour la création d'une classe, au lieu de choisir "Class" choisissez plutôt "PTjava Class".

et bonne programmation parallèle ...

IV. présentation de l'application détection de contours d'image.

Il s'agit d'une implémentation de l'algorithme de Canny pour la détection des contours d'image, l'application se présente comme une fenêtre GUI, permettant à l'utilisateur de sélectionner une image de son disque dure, ensuite appliquez un filtre de détection de contours pour l'image, le traitement peut se faire d'une manière séquentiel, parallèle avec une décomposition des données par blocks ou parallèle avec une décomposition cyclique des données.



Annexe B : codes sources

Listing 1 : invocation des taches parallèles

```
package hello;
import paratask.runtime.*;

public class HelloWorld {



      public static void main(String[] args) {
            System.out.println("(1)");

            hello("Sequential");

            System.out.println("(2)");

            TaskID id1 = task_hello();

            System.out.println("(3)");

            TaskIDGroup id2 = multi_hello();

            System.out.println("(4)");

            TaskID id3 = interactive_hello();

            System.out.println("(5)");

            TaskIDGroup g = new TaskIDGroup(3);
            g.add(id1);
            g.add(id2);
            g.add(id3);

            System.out.println("** Going to wait for the tasks to
execute... ");

            try {
                  g.waitTillFinished();
            } catch (ExecutionException e) {
                  e.printStackTrace();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("** Done! All tasks have now
completed.");
      }

      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");
      }
```

```
      public static void hello(String name) {
//          randomSleep();   //-- make the method take longer
            System.out.println("Hello from "+name);
      }

      public static void randomSleep() {
            try {
                  Thread.sleep((int)(Math.random()*100));
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

Listing 2 : différence entre les types de taches

```
package wiki;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.util.concurrent.ConcurrentLinkedQueue;

public class WebAccess {

public static void webAccess(String address) {
try {
long start = System.currentTimeMillis();
            URL url = new URL(address);
            URLConnection con = url.openConnection();
con.setDoInput(true);
            InputStream is = con.getInputStream();
            BufferedReader bf = new BufferedReader(new
InputStreamReader(is));
            String line = "";
int count = 0;
while ((line = bf.readLine()) != null) {
if (line.contains("a href")) count++;
            }
long time = System.currentTimeMillis() - start;
long id = Thread.currentThread().getId();
            String item =
address.substring(address.lastIndexOf('/')+1);
System.out.println(count + " links for \'" + item+"\' [" + (time /
1000.0) + " seconds, thread "+id+"]");
        } catch (Exception e) {
e.printStackTrace();
        }
    }

    TASK public static void webAccessTask(String address) {
      webAccess(address);
    }
```

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

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

      public static void main(String[] args) {
            ConcurrentLinkedQueue<String> urls = new
ConcurrentLinkedQueue<String>();
urls.add("http://en.wikipedia.org/wiki/New_Zealand");
urls.add("http://en.wikipedia.org/wiki/Auckland");
urls.add("http://en.wikipedia.org/wiki/Rotorua");
urls.add("http://en.wikipedia.org/wiki/Milford_Sound");
urls.add("http://en.wikipedia.org/wiki/Christchurch");
urls.add("http://en.wikipedia.org/wiki/Dunedin");
urls.add("http://en.wikipedia.org/wiki/Queenstown,_New_Zealand");
urls.add("http://en.wikipedia.org/wiki/Tongariro_National_Park");

            Case run = Case.Sequential;
//          Case run = Case.Tasks;
//          Case run = Case.MultiTasks;
//          Case run = Case.InteractiveTasks;

            TaskIDGroup g;

            long start = System.currentTimeMillis();

            switch (run) {
            case Sequential:
                  for(String s: urls) {
                  webAccess(s);
            }
                  break;
            case Tasks:
                  g = new TaskIDGroup(urls.size());
                  for(String s: urls) {
                  TaskID id = webAccessTask(s);
                  g.add(id);
            }
                  try {
                        g.waitTillFinished();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
                  break;

            case MultiTasks:
                  g = webAccessMulti(urls);
                  try {
                        g.waitTillFinished();
                  } catch (Exception e) {
                        e.printStackTrace();
```

```
                        }
                        break;
                  case InteractiveTasks:
                        g = new TaskIDGroup(urls.size());
                        for(String s: urls) {
                        TaskID id = webAccessInteractiveTask(s);
                        g.add(id);
                  }
                        try {
                              g.waitTillFinished();
                        } catch (Exception e) {
                              e.printStackTrace();
                        }
                        break;
                  }

      long time = System.currentTimeMillis() - start;
      System.out.println("Total time: "+(time/1000.0)+" seconds.");
            }

      private enum Case {Sequential, Tasks, MultiTasks,
      InteractiveTasks};
      }
```

Listing 3: détection des contours d'image

- *Classe **contour** fait le traitement de détéction des contourscontient deux methodes**niveau_de_gris** et **canny***

```
package contour;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.*;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import javax.imageio.*;
import paratask.runtime.*;

class Contour {


     public static void  niveau_de_gris(BufferedImage img1,BufferedImage
                                          img2, int debut, int fin){
                  int w = img1.getWidth(null);
                  int h = img1.getHeight(null);

                  for (int i = 0; i < w; i++) {
                        for (int j = debut; j < fin; j++) {
                        Color pixelcouleur= new Color(img1.getRGB(i,j));
                              int R = pixelcouleur.getRed();
                              int G = pixelcouleur.getGreen();
```

```
                int B = pixelcouleur.getBlue();
                int NG=(R+G+B)/3;
                int rgb=new Color(NG,NG,NG).getRGB();
          // changer la couleur de pixel avec la nouvelle couleur
                                                          inversée
                img2.setRGB(i, j, rgb);

                }
          }

          //return img2;

}


     public static void Canny(BufferedImage img,BufferedImage
                                  img2, int debut, int fin){
          int w = img.getWidth();
          int h = img.getHeight();
          int [][] pixel= new int[w][h];
          int x,y,g;
          niveau_de_gris(img,img,debut,fin);


          for (int i = 0; i < img.getWidth(); i++) {
                for (int j = debut; j < fin; j++) {
                      pixel[i][j]=img.getRGB(i, j);
                }
          }
          for (int i = 1; i < img.getWidth()-2; i++) {
                for (int j = debut+1; j < fin-2; j++) {

                      x=-pixel[i][j]+pixel[i][j+2];
                      y=pixel[i][j]-pixel[i+2][j];
                      g=Math.abs(x)+Math.abs(y);

                      pixel[i][j]=g;

                            // changer la couleur de pixel avec la
                                        nouvelle couleur inversée
                      img2.setRGB(i, j, g);
                }
          }


          }

}
```

- *Classe **interface** permet la visualisation de l'application GUI et fait appele aux différents traitements :*

```
package contour;

import java.awt.Dimension;
import java.awt.Event;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;

import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.KeyStroke;
import javax.swing.SwingUtilities;

import paratask.runtime.*;



public class intrface extends JFrame{
	private JMenu jMenuFichier = null;
	private JMenuItem jMenuItemparallel=null;
	private JMenuItem jMenuItemparallel2=null;
	private JMenuItem jMenuItemsequential=null;
	private JMenuItem jMenuItemOuvrir = null;
	private JMenuBar jJMenuBar = null;
	private JPanel jContentPane = null;
	JLabel compte=new JLabel("second");
	private JButton saisie = new JButton ("CHOIX") ;

	private BufferedImage img;
	private static BufferedImage img2;
	//private static Contour contour;
	private MyImage fenetereActuelle;

	private JMenuBar getJJMenuBar() {
		if (jJMenuBar == null) {
			jJMenuBar = new JMenuBar();
			jJMenuBar.setPreferredSize(new Dimension(10, 20));
			jJMenuBar.add(getJMenuFichier());

		}
		return jJMenuBar;
	}

	private JMenu getJMenuFichier() {
		if (jMenuFichier == null) {
			jMenuFichier = new JMenu();
			jMenuFichier.setName("Fichier");
```

```
                jMenuFichier.setPreferredSize(new Dimension(50, 13));
                jMenuFichier.setText("Fichier");
                jMenuFichier.add(getJMenuItemOuvrir());
                jMenuFichier.add(getJMenuItemsequential());
                jMenuFichier.add(getJMenuItemparallel());
                jMenuFichier.add(getJMenuItemparallel2());


          }
          return jMenuFichier;
     }
     private JMenuItem getJMenuItemOuvrir() {
          if (jMenuItemOuvrir == null) {
                jMenuItemOuvrir = new JMenuItem();
                jMenuItemOuvrir.setText("Ouvrir");

     jMenuItemOuvrir.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_
O, Event.CTRL_MASK, false));
                jMenuItemOuvrir.setName("Ouvrir");
                jMenuItemOuvrir.addActionListener(new
java.awt.event.ActionListener() {
                      public void
actionPerformed(java.awt.event.ActionEvent e) {
                            System.out.println("actionPerformed()"); //
TODO Auto-generated Event stub actionPerformed()
                            File f = FilePreview.main(null);

                            affecteimage(f);

                            fenetereActuelle = new MyImage(img);



                            //PanelImage.paintComponent(Graphics g);

                      }
                });
          }
          return jMenuItemOuvrir;
     }

     private JMenuItem getJMenuItemparallel() {
          if (jMenuItemparallel == null) {
                jMenuItemparallel = new JMenuItem();
                jMenuItemparallel.setText("BLOCK parallel processing");
                jMenuItemparallel.setName("BLOCK parallel
processing");
                jMenuItemparallel.addActionListener(new
java.awt.event.ActionListener() {
                      public void
actionPerformed(java.awt.event.ActionEvent e) {
                            System.out.println("actionPerformed()"); //
TODO Auto-generated Event stub actionPerformed()

                            int w = img.getWidth(null);
                            int h = img.getHeight(null);
```

```
                              BufferedImage imgr=new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);;


                              long start = System.currentTimeMillis();

                              BlockParallel(img,imgr);
                        long time = System.currentTimeMillis() - start;
                  System.out.println("block: "+time/ 1000.0 +"seconds" );


                              fenetereActuelle = new MyImage(imgr);

                        }
                  });
            }
            return jMenuItemparallel;
      }

      private JMenuItem getJMenuItemparallel2() {
            if (jMenuItemparallel2 == null) {
                  jMenuItemparallel2 = new JMenuItem();
                  jMenuItemparallel2.setText("CYCLIC parallel
processing");
                  jMenuItemparallel2.setName("CYCLIC parallel
processing");
                  jMenuItemparallel2.addActionListener(new
java.awt.event.ActionListener() {
                        public void
actionPerformed(java.awt.event.ActionEvent e) {
                              System.out.println("actionPerformed()"); //
TODO Auto-generated Event stub actionPerformed()

                              int w = img.getWidth(null);
                              int h = img.getHeight(null);
                              BufferedImage imgr=new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);;

                              long start = System.currentTimeMillis();

                              CyclicParallel(img,imgr);
                        long time = System.currentTimeMillis() - start;
            System.out.println("cyclic: "+time/ 1000.0 +"seconds" );


                              fenetereActuelle = new MyImage(imgr);



                        }
                  });
            }
            return jMenuItemparallel2;
      }

      private JMenuItem getJMenuItemsequential() {
            if (jMenuItemsequential == null) {
                  jMenuItemsequential = new JMenuItem();
```

```
                jMenuItemsequential.setText("sequential processing");
                jMenuItemsequential.setName("sequential processing");
                jMenuItemsequential.addActionListener(new
java.awt.event.ActionListener() {
                      public void
actionPerformed(java.awt.event.ActionEvent e) {
                           System.out.println("actionPerformed()"); //
TODO Auto-generated Event stub actionPerformed()

                           int w = img.getWidth(null);
                           int h = img.getHeight(null);
                           BufferedImage imgr=new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);;


                           long start = System.currentTimeMillis();
                           Contour.Canny(img,imgr,0,h);
                           long time = System.currentTimeMillis() -
start;
                           System.out.println("sequentiel: "+time/
1000.0 +"seconds" );
                            //nvge(img,imgresultat);
                           //imgresultat=Contour.nvg( img);


                           fenetereActuelle = new MyImage(imgr);



                      }
                });
          }
          return jMenuItemsequential;
     }

     public static void main(String[] args) {
          // TODO Auto-generated method stub
          SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                      intrface thisClass = new intrface();

     thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                      thisClass.setVisible(true);
                }
          });
     }

     public intrface() {
          super();
          initialize();
     }

     /**
      * This method initializes this
      *
      * @return void
      */
     private void initialize() {
```

```
            this.setSize(400,300);
            this.setLocation(new Point(200, 300));
            this.setJMenuBar(getJJMenuBar());
            add(saisie);
            this.setContentPane(getJContentPane());

            this.setTitle("Bastos Image Processing");
            this.setVisible(true);
      }

      // affectation de l'image

      void affecteimage(File f)
      {

            try {

                  this.img = ImageIO.read(f);
            } catch (IOException e) {
                  return;
            }
      }
      /*void nvge (BufferedImage img1, BufferedImage img2)
      {
            Contour.nvg( img1, img2);
      }*/

      public static void BlockParallel(BufferedImage img,BufferedImage
imgr)
       {
            int w = img.getWidth(null);
                  int h = img.getHeight(null);

              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);

                  try {
                        g.waitTillFinished();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
       }

      public static void CyclicParallel(BufferedImage img,BufferedImage
imgr)
       {
            int w = img.getWidth(null);
                  int h = img.getHeight(null);
                  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);


                        try {
                              g.waitTillFinished();
                        } catch (Exception e) {
                              e.printStackTrace();
                        }
        }


        TASK public static void taskgris(BufferedImage img,BufferedImage
imgr, int x, int y)
       {

             long start = System.currentTimeMillis();

long id = Thread.currentThread().getId();
             Contour.Canny(img,imgr,x,y);
             long time = System.currentTimeMillis() - start;
                   System.out.println(time / 1000.0 + " seconds, thread
["+id+"]");
                    }

        TASK public static void 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;


             Contour.Canny(img,imgr,debut,fin);

              }
                    }

       private JPanel getJContentPane() {
             if (jContentPane == null) {
                   jContentPane = new JPanel();

                   jContentPane.setLayout(null);


                   jContentPane.setPreferredSize(new Dimension(1, 1));
             }
             return jContentPane;
       }

}
```

- *Classe **myImage** permet l'affichage des images :*

```
package contour;


import java.awt.Color;
import java.awt.Dimension;
import java.awt.Event;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.*;


public class MyImage extends JFrame {


private paneau pan;
BufferedImage img;

public MyImage (BufferedImage img)
{


      this.img=img;
      paneau p = new paneau(img);
      this.setContentPane(p);
      this.setSize(550,550);
      this.setVisible(true);
            this.setTitle("Bastos Image Processing");
}


public class paneau extends JPanel {
      BufferedImage imgp;

      paneau(BufferedImage imge) {


      this.imgp=imge;
      }

      public void paintComponent(Graphics g){

            //Vous pourrez voir cette phrase à chaque fois que la
            //méthode est invoquée !
                  g.drawImage(imgp, 0, 0, null)
            }

public void setImg(BufferedImage img) {
      this.imgp=img;
      this.repaint();
      }

}
public void setImg(BufferedImage img) {
      this.img=img;
```

```
      pan.setImg(img);

}
public BufferedImage getImg() {
      return this.img;
}
            }
```

Bibliographie

[1]Braunl , T.(1993) Parallel programming an introduction. London : Printice Hall

[2] Deghilage,G.(1992) Architecture et programmation parallèle : approche pratique en environement scientifique sur multiprocesseurs. Paris : Addisson Wesly

[3] Arnold,A.(1992)Programmation parallèle : modèles et validation. Paris : Armand Colui

[4] Danny Dig, A Refactoring Approach to Parallelism. PARALLELISM ON THE DESKTOP, IEEE 2011, vol 11, p 17-22 .

[5] Wooyoung Kim et Michael Voss, Intel Multicore Desktop Programming with Intel Threading Building Blocks. PARALLELISM ON THE DESKTOP.IEEE 2011 vol 11, p 23-31.

[6]Ami Marowkac, Think Parallel: Teaching Parallel Programming Today, (vol. 9, no. 8). no. 0808, IEEE 2008

[7] Victor Pankratius, Wolframe Schulte et Kurt Keutzer, PARALLELISM ON THE DESKTOP, IEEE 2011 vol 11, p 14-16.

[8]Nasser Giacaman et Oliver Sinnen**,** Object-Oriented Parallelization of Java Desktop Programs. PARALLELISM ON THE DESKTOP.IEEE 2011 vol 11, p 32-38.

[9] Nasser Giacaman et Oliver Sinnen,Task Parallelism for Object Oriented Programs.The International Symposium on Parallel Architectures, Algorithms, and Networks.IEEE 2008 vol 08, p 13-18.

Webographie

[10]ThoughtWorks, Refactoring. [en ligne]. *http://refactoring.com/* [page consultée le 14/12/2010].

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

[12] *Ian Foster, Designing and Building Parallel Programs [en ligne] http://www.mcs.anl.gov/~itf/dbpp/*

[13] Blaise Barney, Introduction to Parallel Computing.[en ligne]. *https://computing.llnl.gov/tutorials/parallel_comp/* [page consulté le 03/04/2010]