École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
3e année
2013 - 2014

Rapport de Projet - Systémes d'Exploitation

Programmation multi-coeur C# .NET pour le traitement sous AForge.NET

Encadrant
Mathieu DELALANDRE
mathieu.delalandre@univ-tours.fr

Université François-Rabelais, Tours

étudiants
Horacio CACHINHO
horacio.cachinho@etu.univ-tours.fr
Quentin BOUVIER
quentin.bouvier@etu.univ-tours.fr

DI3 2013 - 2014

Version du 9 juin 2014

Table des matières

Table des figures

Liste des tableaux

Symboles et Abréviations

bbp Bits per pixel

FFT Fast Fourier Transform (Transformée de Fourier Rapide)

RoI Region of Interest (Région d'intérêt)

TPL Task Parallel Library

Introduction

De nos jours, la plus part des processeurs existants sur le marché, comportent 4 coeurs, les plus performants pouvant même atteindre jusqu'à 8 voir 16 coeurs. Ainsi, lorsque l'on effectue des traitements nécessitant beaucoup de ressources système, le calcul multi-coeur nous permet de mieux gérer ces ressources afin d'optimiser les temps de traitement.

Supposons quatre traitements, A, B, C et D. Le framework .NET nous permet d'effectuer ces traitements de deux manières différentes :

Séquentiel : B est effectué après A, C après B, D après C.

Parallèle : Dépendant des ressources disponibles, ces quatre traitements sont effectués quasiment en même temps.

Grâce à cette avancée technologique, les calculs auparavant longs et complexes à résoudre, deviennent plus simples et rapides à traiter. La programmation parallèle devient ainsi de plus en plus utilisée dans des applications de calcul ou simulation. Un des domaines pour lesquels on utilise la technologie multi-coeurs est, par exemple, le traitement d'images et les méthodes spectrales comme la Transformée de Fourier Rapide (ou FFT).

Notre projet va donc s'intéresser à la mise en place de la programmation parallèle en utilisant le framework .NET, et se déroulera en partenariat avec la société Itesoft dans le cadre du projet industriel DOD. Nous utiliserons également la librairie AForge.NET, qui nous fournira les algorithmes FFT, permettant *"la mise en place de techniques dites de « template matching »"*. En effet, ces algorithmes ayant des temps de calcul assez grands dû à leurs complexité, nous pourrons tirer profit de la programmation multi-coeur afin de diminuer ces temps de calcul et optimiser le traitement.

Nous commencerons donc par présenter la librairie AForge.Net, qui nous fournira les algorithmes de calcul de FFT, puis, nous poursuivrons par la mise en place de calculs parallèles exploitant la technologie multi-coeur et nous finirons par aborder le problème de la synchronisation des résultats lors de traitements utilisant cette technologie.

AForge.Net

Nous avons pour but de mettre en place un système de calcul multi-coeur se basant sur le *"template matching"* et le calcul de FFT sur une image donnée. Ainsi, afin de concentrer nos efforts sur les méthodes de parallélisation et la programmation multi-coeur, nous avons utilisé une librairie externe nous proposant notamment des algorithmes FFT, nécessaires au projet.

Nous utiliserons donc AForge .NET, une librairie *OpenSource* sous licence LGLPLv3 et en partie GPLv3, crée par Andrew Kirillov et développé en C# et .Net.

Au fur et à mesure de nos recherches sur cette librairie, nous avons pu remarquer l'étendue des utilisations possibles de celle-ci. En effet, cette dernière propose des fonctionnalités utiles à divers domaines tels que :

- Traitements d'images
- Vision par ordinateur
- Simulation de réseaux neuronales
- Reconnaissance des formes

Lors de la réalisation de ce projet, nous avons utilisé la sous-libraire *AForge.Image*, pour avoir accès à toutes les fonctions de calculs de FFT.

1ère Partie : Parallélisation de calculs de FFT

La première partie de notre projet consistait à mettre en place une parallélisation de calculs FFT sur une base d'images donnée. Ces images sont des RoI appartenant à une image de base.

En effet, on est capable d'extraire certaines parties d'une image (des régions d'intérêt) qui sont ensuite nécessaires à de nombreux traitements. Nous pouvons prendre comme exemple une facture ayant pour régions d'intérêt un logo, une adresse ou bien les quantités de produits commandés. Grâce aux techniques dites de *"template matching"* et de ces RoI, les factures de notre exemple peuvent être traitées automatiquement et donc plus rapidement.

Cette technique de *"template matching"* compare une RoI à une base d'images pour en extraire celle qui ressemble le plus.

4.1 Le calcul multi-coeur

Tout d'abord, nous avons dû étudier le framework .NET ainsi que les fonctionnalités qu'il nous propose pour la parallélisation des traitements. La version 4+ de ce framework nous propose une librairie appellée TPL qui facilite l'implémentation d'algorithmes se servant du multi-threading et du parallélisme. Nous allons donc utiliser les fonctionnalités proposées par cette librairie tout au long de notre projet.

Les fonctionnalités qui nous intéresseront le plus sont les suivantes :

Task : La classe *Task* représente une opération asynchrone

Parallel.ForEach : Exécute en parallèle les itérations de la boucle *foreach*

Parallel.For : Exécute en parallèle les itérations de la boucle *for*

MaxDegreeOfParallelism : Appartenant à la classe *ParallelOptions*, permet de définir le niveau de parallélisation du traitement à effectuer.

Pour tester ces fonctionnalités, et dans le cadre de notre premier objectif, c'est à dire, l'implémentation du calcul multi-coeur sous .NET, nous avons mis en place un logiciel qui permet de comparer les temps mis pour effectuer une multiplication entre deux matrices de taille 2^n, en utilisant, d'un coté, une méthode de calcul séquentiel, et d'un autre côté, une méthode de calcul parallèle.

Voici l'algorithme qui effectue la multiplication de deux matrices de taille n :

Multiplication de deux matrices de taille n

1: $matrix1 = GenerateRandomMatrix(n, n)$
2: $matrix2 = GenerateRandomMatrix(n, n)$
3: **pour** $i = 0$ to n **faire**
4: **pour** $j = 0$ to n **faire**
5: **pour** $k = 0$ to n **faire**
6: $matrixResult[i, j] \leftarrow matrixResult[i, j] + (matrix1[i, k] \times matrix2[k, j])$
7: **fin pour**
8: **fin pour**
9: **fin pour**

Nous avons donc crée deux méthodes, basées sur cet algorithme. La première effectue les multiplications de manière séquentielle (utilisant donc les boucles *for* normales), la deuxième implémentant la boucle *Parallel.For* fournie par la TPL.

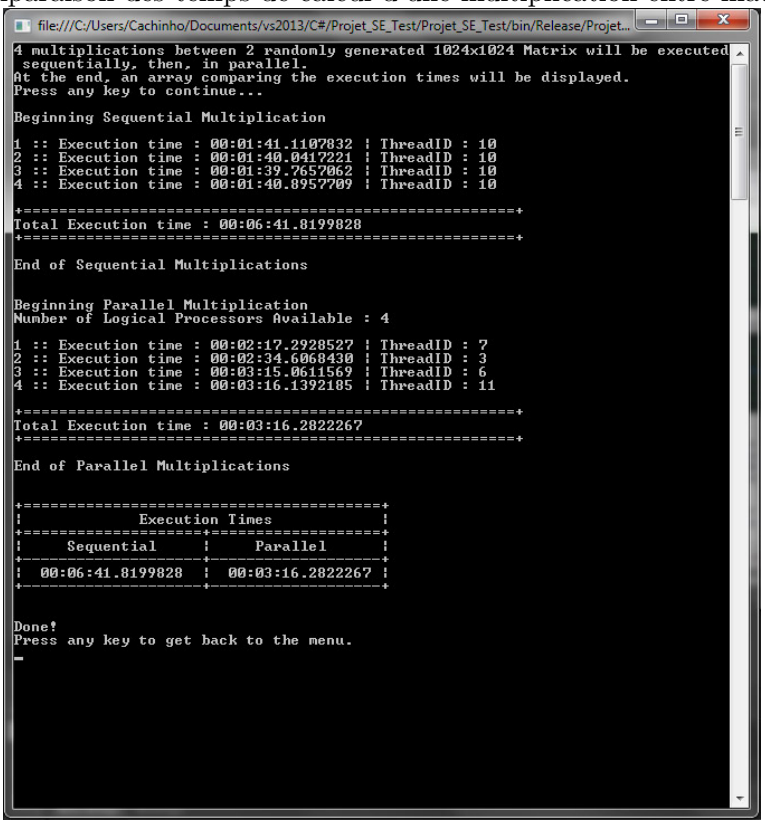
Voici ci-dessous ces deux méthodes :

```
1  //Méthode séquentielle
2  for (int i = 0; i < n; i++)
3  {
4     for (int j = 0; j < n; j++)
5     {
6        for (int k = 0; k < n; k++)
7        {
8           matrixResult[i, j] += matrix1[i, k] * matrix2[k, j];
9        }
10    }
11 }
```

```
1  //Méthode parallèle
2  Parallel.For(0, n, i =>
3  {
4     for (int j = 0; j < n; j++)
5     {
6        for (int k = 0; k < n; k++)
7        {
8           matrixResult[i, j] += matrix1[i, k] * matrix2[k, j];
9        }
10    }
11 });
```

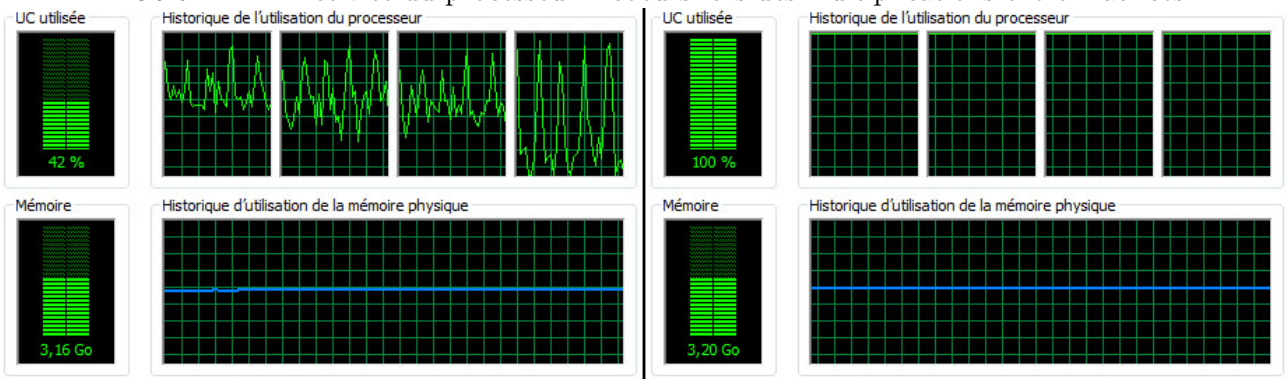
Les résultats de l'implémentation de ces deux méthodes sur un processeur 4 coeurs :

FIGURE 4.1 – Comparaison des temps de calcul d'une multiplication entre matrices sous 4 coeurs



Ici nous avons effectué quatre multiplications en utilisant la méthode séquentiel puis, quatre multiplications en utilisant la méthode parallèle. Comme nous pouvons remarquer, les quatre multiplications effectuées en séquentiel, utilisent la même *thread* et le calcul dure, au total, 6 minutes et 41 secondes. Quant à la méthode parallèle, chaque multiplication est affectée à une *thread* différente. Les traitements seront ainsi effectuées de manière asynchrone et le calcul ne durera que 3 minutes et 16 secondes au total. Ci-dessous, l'activité du processeur pendant ces calculs, à gauche pendant le calcul séquentiel, à droite, pendant le calcul parallèle :

FIGURE 4.2 – Activité du processeur 4 coeurs lors des multiplications entre matrices



Nous remarquons que la méthode parallèle, contrairement à la méthode séquentielle, utilise toutes les ressources processeur disponibles.

Nous avons également testé ces méthodes sur un processeur plus puissant (8 coeurs cadencés à 4.2Ghz). Voici les résultats obtenus :

FIGURE 4.3 – Comparaison des temps de calcul d'une multiplication entre matrices sous 8 coeurs

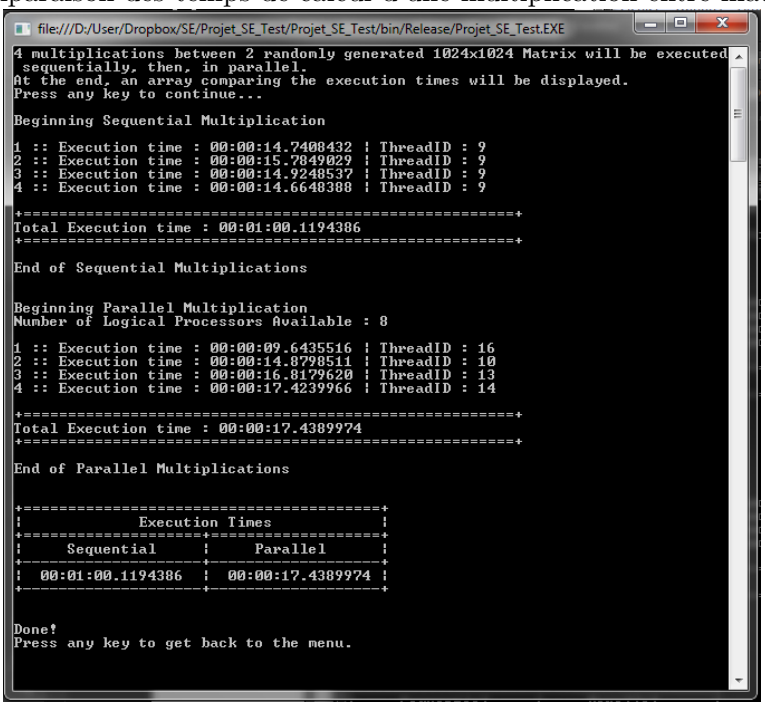
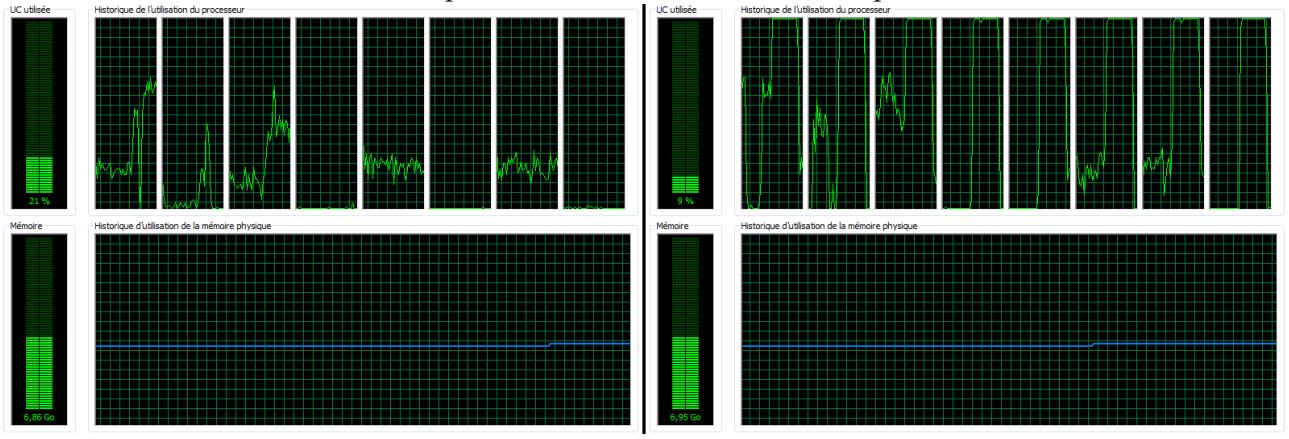


FIGURE 4.4 – Activité du processeur 8 coeurs lors des multiplications entre matrices



Nous remarquons que les temps de calcul sont nettement plus faibles.

Maintenant que nous avons trouvé comment utiliser la TPL pour implémenter des calculs multi-coeur, nous pouvons passer à l'étape suivante : l'implémentation de la FFT.

4.2 Temps de calcul de la FFT

Notre objectif, dans cette partie de notre projet, est d'ajouter l'algorithme de calcul de la FFT, fourni par AForge .NET, à notre méthode de calcul multi-coeur.

Tout d'abord, nous avons du étudier la documentation d'Aforge afin de trouver comment utiliser l'algorithme. Après quelques recherches et essais nous avons abouti à la méthode suivante :

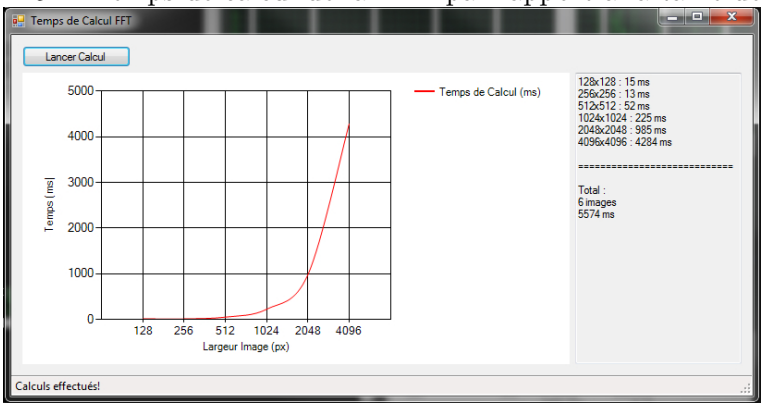
```
1  private void FFT(Bitmap image)
2  {
3      try
4      {
5          // create complex image
6          ComplexImage complexImage = ComplexImage.FromBitmap(image);
7          // do forward Fourier transformation
8          complexImage.ForwardFourierTransform();
9
10         // get complex image as bitmat
11         Bitmap fourierImage = complexImage.ToBitmap();
12     }
13     catch (Exception ex)
14     {
15         //Display and handle the exception
16     }
17 }
```

Cependant, afin de pouvoir utiliser cette méthode, nous devons préalablement charger la RoI à traiter en mémoire et effectuer des vérifications / traitements sur celle-ci. En effet, l'algorithme FFT d'Aforge .NET ne fonctionne qu'avec des images en niveaux de gris de type 8bbp indexed et de taille 2^n.

M.Delalandre, notre encadrant, nous a donc fourni des images de test correspondant à ces critères afin qu'on puisse tester nos algorithmes.

Nous avons ainsi abouti à un algorithme qui nous permet d'estimer le temps nécessaire au calcul de la FFT sur une image donnée. Voici les temps de calcul de la FFT, par rapport à la taille de l'image, obtenu sur un processeur 8 coeurs.

FIGURE 4.5 – Temps de calcul de la FFT par rapport à la taille de l'image



Nous avons également effectué cette mesure sur le processeur 4 coeurs. Voici les résultats obtenus :

128x128 66ms

256x256 42ms

512x512 175ms

1024x1024 752ms

2048x2048 3160ms

4096x4096 13876ms

Nous pouvons remarquer que les temps de calcul forment une courbe de type exponentiel. Ainsi, nous pouvons conclure qu'une image de taille 2^{n+1} aura un temps de calcul d'environ 4.2 fois supérieur à une image de taille 2^n.

2ème Partie : Synchronisation multi-coeur

Dans la deuxième partie de notre projet, nous avons du étendre les fonctionnalités implémentées dans les parties précédentes à une base de RoI réelle et réfléchir au problème de la synchronisation des résultats.

5.1 La préparation de la base de RoI

Lorsqu'on a reçu la base de RoI, contenant plus de 5600 images, la première chose que nous avons remarqué c'est la quantité d'images à traiter. Nous avons donc dû chercher une nouvelle méthode de chargement des RoI à traiter car la méthode implémentée auparavant ne marchait plus. En effet, comme nous n'avions que 6 images à traiter, nous chargions les images en tant que *Bitmap* directement en mémoire afin d'éviter les temps de chargement intermédiaires. Cependant, cette méthode n'est plus valable pour les $5600+$ RoI à traiter. De plus, nous avons également remarqué que ces RoI n'étaient plus de taille 2^n

Pour répondre à la première problématique (donc, le chargement des images), nous avons pensé aux structures de données de type dictionnaire, proposées par le langage C#. Notre but était de charger en mémoire les noms des fichiers à traiter, puis de les utiliser pour charger le fichier lorsqu'il doit être traité. Cela nous permettrait d'optimiser l'espace occupé en mémoire. Cependant, comme cette nouvelle méthode requiert des chargements intermédiaires, nous avons remarqué que les temps de calcul seraient faussés par ces chargements, et nous avons dû également changer cela. Nous faisons désormais appel à la classe *Stopwatch* qui encapsule l'instantiation *Task* qui se chargera du calcul FFT de la RoI en question, afin d'obtenir des résultats encore plus précis. Voici l'implémentation :

```
1  private double MesureFFT(Bitmap image)
2  {
3      double elapsed = 0;
4
5      try
6      {
7          var stopwatch = new Stopwatch();
8          stopwatch.Start();
9
10         Task task = Task.Factory.StartNew(() => FFT(image));
11         while (!task.IsCompleted)
12         {
13             task.Wait(); //wait for the task to be completed
14         }
15
16         stopwatch.Stop();
17         elapsed = stopwatch.ElapsedMilliseconds;
18     }
19     catch (Exception ex)
20     {
21         //Display and handle the exception
22     }
23
24     return elapsed;
25 }
```

Afin de répondre à la deuxième problématique (les images de taille différente à 2^n), nous avons dû développer et implémenter un algorithme de padding. Cet algorithme prends en entrée une image de type *Bitmap* et ajoute des pixels d'une couleur donnée au tour de l'image de base jusqu'à ce que celle-ci ait une taille carrée de coté 2^n. Voici l'algorithme développé et son implémentation en C# :

Ajout de bordures

1: $powSup \leftarrow$ "Puissance de 2 supérieure à la taille de l'image"
2: $newImage \leftarrow$ "Nouvelle image de taille powSup x powSup"
3:
4: $/*$ On calcule le centre ou la RoI doit être placée $*/$
5: $x \leftarrow (powSup - RoI.Largeur)/2$
6: $y \leftarrow (powSup - RoI.Hauteur)/2$
7:
8: $/*$ On ajoute la RoI au centre de newImage $*/$
9: $/*$ On colorie les pixels de padding de la couleur désirée $*/$

Cependant, afin de pouvoir implémenter cet algorithme, nous avons du préalablement calculer la puissance de 2 supérieure la plus proche du plus grand coté de la RoI. Nous avons ainsi abouti aux trois méthodes suivantes :

```
1  private int GetMax(Bitmap image)
2  {
3      return image.Height <= image.Width ? image.Width : image.Height;
4  }
```

```
1  private int Find2Pow(int value)
2  {
3      double pos = Math.Ceiling(Math.Log(value, 2));
4      return (int)(Math.Pow(2, pos));
5  }
```

```
1  private Bitmap AddPadding(Bitmap image, Color paddingColor)
2  {
3      int pow = Find2Pow(GetMax(image));
4      Bitmap newImage = new Bitmap(pow, pow);
5      try
6      {
7          using (Graphics graphics = Graphics.FromImage(newImage))
8          {
9              graphics.Clear(paddingColor);
10             int x = (int)((pow - image.Width) / 2);
11             int y = (int)((pow - image.Height) / 2);
12             graphics.DrawImage(image, x, y, image.Width, image.Height);
13
14             newImage = newImage.Clone(new Rectangle(0, 0, newImage.Width, newImage.
                   Height), PixelFormat.Format8bppIndexed);
15         }
16     }
17     catch (Exception ex)
18     {
19         //Display and handle the exception
20     }
21
22     return newImage;
23 }
```

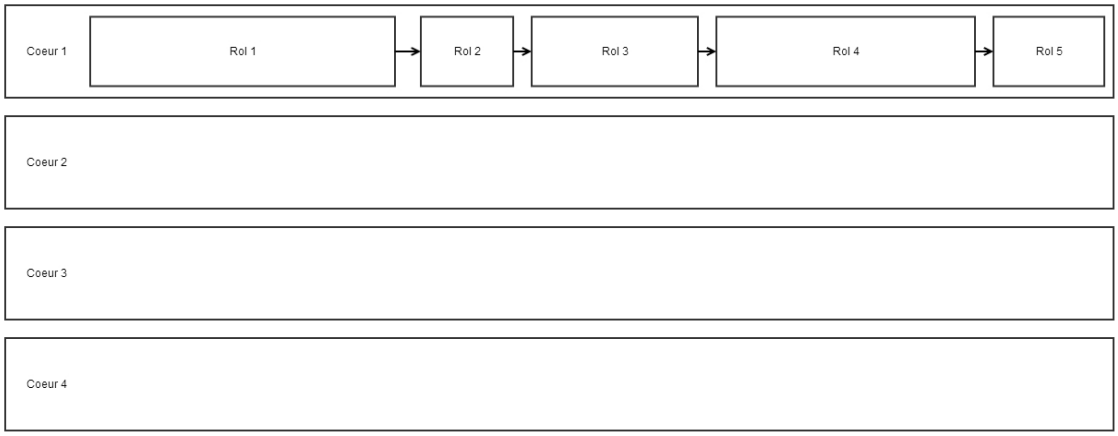
La première méthode (*GetMax*) nous permet de trouver la valeur du coté le plus grand de la RoI, la deuxième méthode (*Find2Pow*) nous permet de calculer la puissance de 2 la plus proche et, enfin, la troisième méthode (*AddPadding*) nous sert à ajouter les bordures afin que la RoI puisse être exploitable par l'algorithme de calcul de la FFT.

5.2 L'algorithme de distribution des tâches

Une fois que nous avons trouvé une solution aux problématiques rencontrées, nous avons commencé à effectuer des recherches sur la façon dont nous pourrions optimiser le traitement de ces 5600+ RoI de notre base. Comme nous avons remarqué lors de la détermination des temps de calcul de la FFT, une image de taille 2^{n+1} nécessite un temps de calcul d'environ 4.2 fois supérieur à une image de taille 2^n.

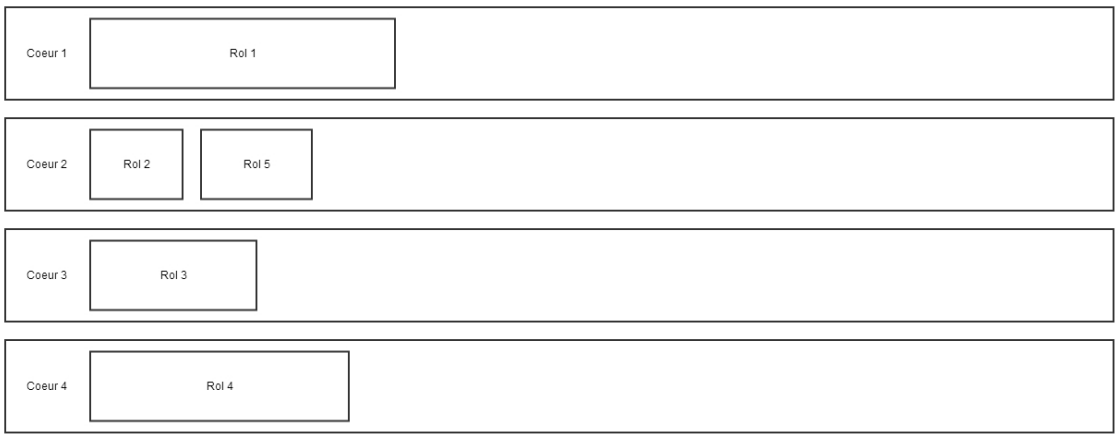
Ainsi, afin d'optimiser notre temps de traitement, la meilleur solution serait de prendre en compte le temps de calcul moyen d'une RoI donnée, et attribuer sont traitement au coeur du processeur dont la queue des tâches à exécuter mettrais le moins de temps a être libérée. Voici un exemple :

FIGURE 5.1 – Schéma de gestion des traitements sous forme séquentielle



Comme nous pouvons voir dans l'exemple ci-dessous, les RoI sont traitées de manière séquentielle. Le temps du traitement de ces cinq RoI serait donc la somme des temps de calcul de chacune de ces RoI.

FIGURE 5.2 – Schéma de gestion des traitements sous forme parallèle



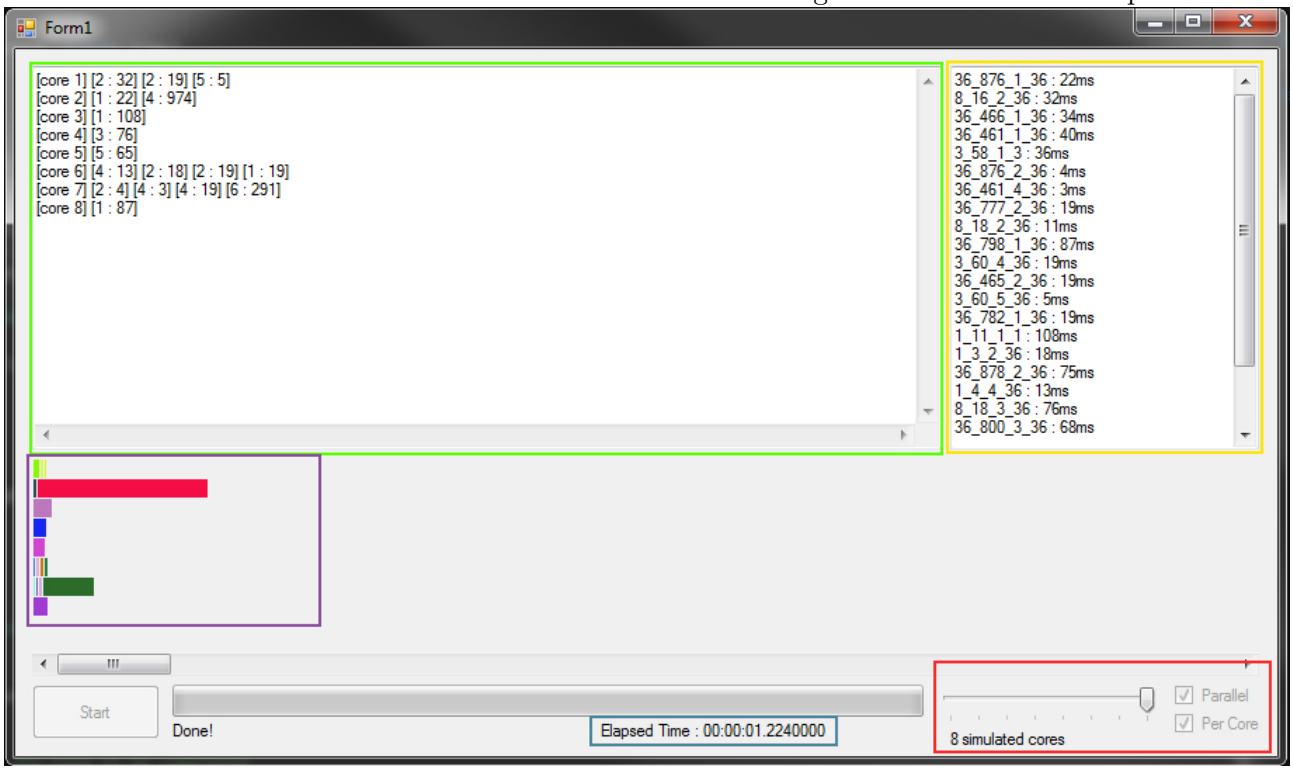
Ci-dessous, l'optimisation du traitement grâce notamment aux fonctionnalités de parallélisation des calculs. Nous pouvons noter que le temps total du traitement n'est plus la somme des temps de calcul de chacune des RoI, mais bien le temps nécessaire au traitement complet de la queue du coeur le plus chargé.

Afin de calculer la distribution des tâches optimale, par rapport au nombre de coeurs disponibles sur le processeur, nous avons du développer un algorithme de distribution des tâches. Cet algorithme fonctionne de la façon suivante :

1. On charge les noms des fichiers dans un tableau, et on détermine le temps de calcul de la FFT correspondant à sa taille.
2. On boucle dans le tableau pour trouver la tâche qui mettrais le plus de temps a s'exécuter.
3. On recherche le coeur du processeur dont la queue d'exécution (les temps qui mettront les taches en attente à s'exécuter) est la moins longue.
4. On insère la tâche dans la queue d'exécution du coeur correspondant et on la supprime du tableau.
5. On revient à l'étape 2 jusqu'à ce qu'il y en ait plus de tâches à distribuer.

Afin d'exploiter cet algorithme, nous avons repris les méthodes développées auparavant et nous avons mis à jour notre logiciel. Voici le résultat de son exécution sur une base de test de 25 images :

FIGURE 5.3 – Résultat d'exécution sur une base de 25 images utilisant la méthode parallèle



Ce logiciel nous détermine le temps de calcul de chaque RoI (zone en jaune), la distribution optimale sous forme de graphique (en violet) et sous forme de tableau (en vert). Nous avons fait en sorte d'obtenir également le temps total de calcul des FFT (en bleu) et nous avons également ajouté des options (en rouge) :

Simulated Cores Une barre numérique qui nous permet de simuler le nombre de coeurs utilisés lors des calculs (cette option fait appel à la fonctionnalité *MaxDegreeOfParallelism* qui nous permet de définir le degré maximal de parallélisme).

Parallel Nous permet de choisir si l'on veut effectuer des calculs utilisant la méthode séquentielle ou multi-coeur

PerCore Nous permet de définir si l'on veut afficher une distribution optimale par coeur disponible, ou bien par image d'appartenance de la RoI.

Ci-dessous quelques résultats obtenus en utilisant différentes configurations :

FIGURE 5.4 – Résultat d'exécution sur une base de 25 images utilisant la méthode séquentielle

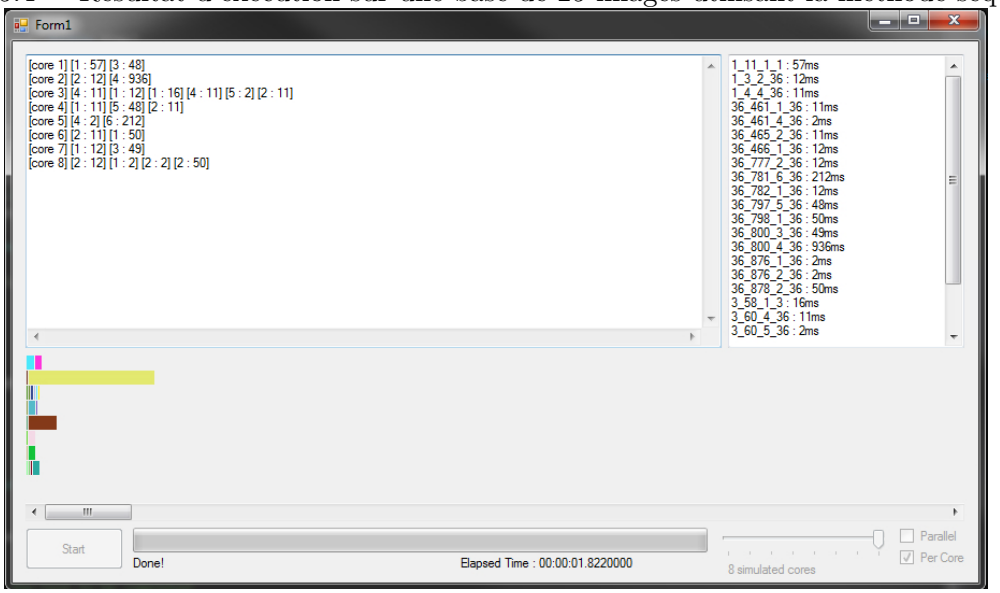
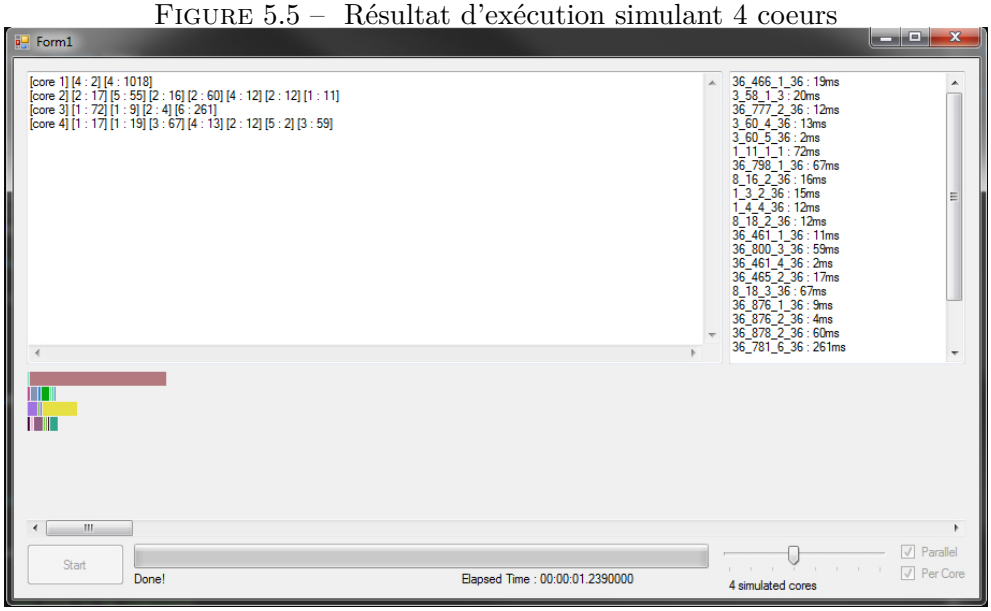


FIGURE 5.5 – Résultat d'exécution simulant 4 coeurs



Ci-dessous quelques résultats obtenus sur la base de RoI :

FIGURE 5.6 – Résultat d'exécution utilisant la méthode parallèle sur la base de RoI

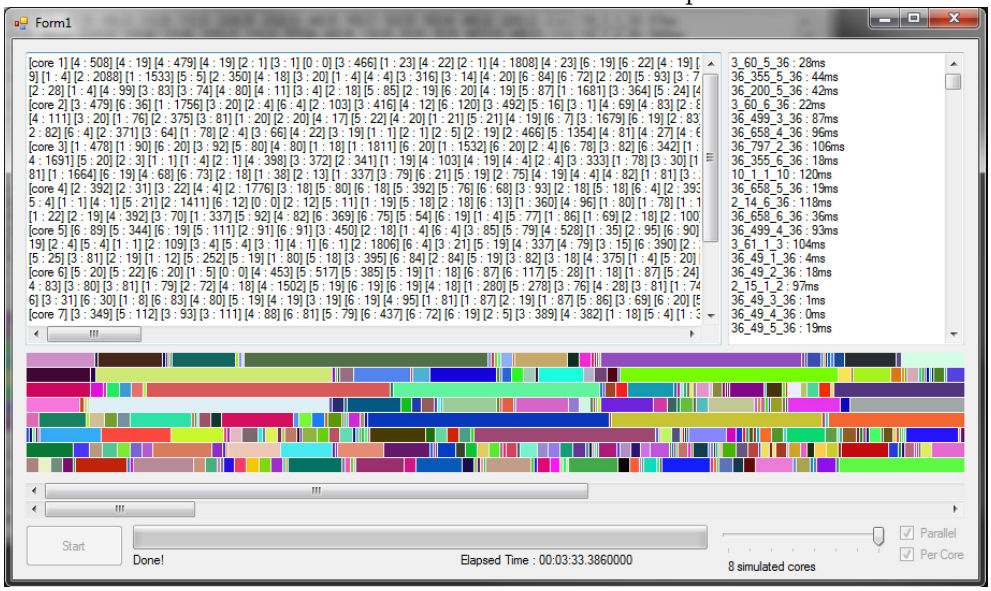
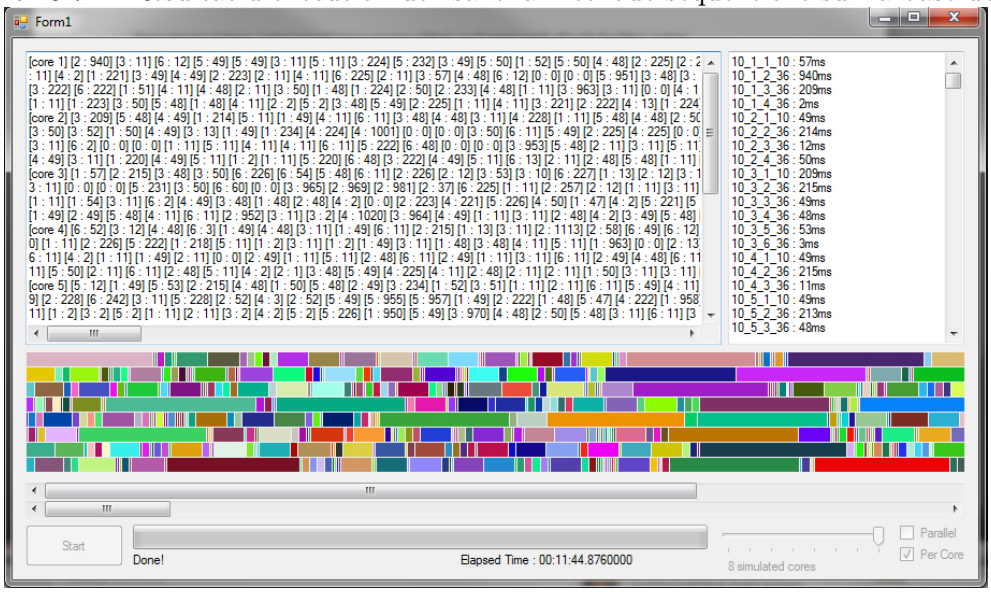


FIGURE 5.7 – Résultat d'exécution utilisant la méthode séquentielle sur la base de RoI



Nous pouvons remarquer que, pour une base d'images assez petite (inférieure à environ 20 images), la différence entre le temps de calcul en mode séquentiel par rapport au mode parallèle est quasi-négligeable. Cependant, pour des bases d'images de taille considérable, les temps de calcul sont nettement différents et la méthode de calcul parallèle est beaucoup plus rapide à s'exécuter :

Séquentiel 11 minutes et 44 secondes

Parallèle 3 minutes et 33 secondes

5.3 La synchronisation des résultats

Le but de cette dernière partie de notre projet, était de réussir à synchroniser les temps de calcul des RoI appartenant à une image donnée afin d'obtenir un temps globale (et par la même occasion, de trouver un moyen de synchroniser des traitements, autres que le temps de calcul).

Cependant, nous n'avons pas abouti à une solution qui nous permettrais d'affecter réellement les tâches aux coeurs physiques du processeur. En effet, la TPL et le système de *Task* gère automatiquement cette attribution. De plus, une tâche n'est pas affectée définitivement à un coeur, c'est à dire, si un autre coeur est moins changé, la TPL changera la tâche vers ce coeur, libérant ainsi du temps de calcul. De plus, nous n'avons pas trouvé un autre moyen de forcer une tâche à être exécutée sur un coeur donné.

Conclusion

Aujourd'hui, le multi-coeur devient quasiment essentiel dans toute application. En effet, les logiciels ont des interfaces graphiques de plus en plus gourmandes, les jeux-vidéo sont de plus en plus détaillés et l'accroissement de la puissance moyenne des composants d'un ordinateur nous permet d'effectuer des calculs, auparavant difficiles et longs à réaliser.

Grâce à ce projet, nous avons pu apprendre les bases de la programmation multi-coeur et mettre en ?uvre les connaissances acquises durant les cours de systèmes d'exploitation, suivis au cours de cette année.

Nous avons également eu l'opportunité d'améliorer nos compétences sur le langage C# et le framework .NET.

Glossaire

Asynchrone Un processus est dit asynchrone lorsqu'il s'exécute simultanément avec le reste du programme. Cela permet la mise en place d'une interface graphique par exemple, lorsque les traitement en arrière-plan s'exécutent simultanément avec l'affichage des résultats à l'écran.

Framework Ensemble de composants logiciels servant de fondations aux logiciels l'utilisant.

Padding Espace qui sépare un élément central de ses bordures externes

Thread Tâche ou Fil d'exécution.

Bibliographie

[1] http://msdn.microsoft.com/en-us/library/618ayhy6.aspx
[2] http://msdn.microsoft.com/fr-fr/library/w0x726c2(v=vs.110).aspx
[3] http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx
[4] http://www.aforgenet.com/

Programmation multi-coeur C# .NET pour le traitement sous AForge.NET

Département Informatique
3e année
2013 - 2014

Rapport de Projet - Systémes d'Exploitation

Résumé : Programmation multi-coeur C# .NET pour le traitement sous AForge.NET

Mots clefs : C#, .NET, multi-coeur, thread, task, fft

Abstract: Description en anglais

Keywords: Mots clès en anglais

Encadrant
Mathieu DELALANDRE
mathieu.delalandre@univ-tours.fr

Université François-Rabelais, Tours

étudiants
Horacio CACHINHO
horacio.cachinho@etu.univ-tours.fr
Quentin BOUVIER
quentin.bouvier@etu.univ-tours.fr

DI3 2013 - 2014