

Introduction à la programmation parallèle

par Lainé Vincent ([autres articles](#))

Date de publication : 27/09/2008

Dernière mise à jour : 27/09/2008

A travers cet article nous allons voir une technique de programmation qui à mon avis est amenée à se répandre: La programmation parallèle.
Pour cela nous utiliserons le framework dotnet et sa nouvelle extension : les "Parallel extensions". La lecture de cet article suppose que vous êtes familier avec la programmation dotnet et surtout la gestion des threads.

I - Introduction.....	3
I-A - Définitions.....	3
I-B - Contexte technologique.....	3
I-C - Threads vs programmation parallèle.....	4
II - Premier programme utilisant les "parallel extensions".....	5
II-A - Premier exemple : Requête select sur une base de données.....	5
II-B - Deuxième exemple : PLinq to Object.....	6
III - Conclusion.....	7
IV - Téléchargements et documentations.....	7
V - Remerciements.....	7

I - Introduction

I-A - Définitions

Avant de nous lancer à corps perdu dans le code, la compilation et le débogage il est nécessaire de faire quelques mises au point du côté définition et technique.

Les "parallel extensions", qui n'ont pas de nom français voulant dire quelque chose, sont un ensemble de classe destiné au framework .NET 3.5. Ces classes sont regroupées dans un seul assembly nommé System.Threading. Le développement de ces "parallel extensions" à l'heure de la rédaction de cet article est au stade de CTP : Community Technology Preview. L'installateur pour la version de juin 2008 est disponible sur le site de **Microsoft en téléchargement**

I-B - Contexte technologique

Pour bien comprendre l'intérêt de la programmation parallèle mais également ces pièges il est nécessaire de savoir se qui cache derrière ces mots.

En temps "normal" lorsque vous programmez vous faites de la programmation dite linéaire. Cette technique consiste à exécuter les instructions les unes après les autres afin d'obtenir le résultat souhaité.

```
static void Mainlineaire()
{
    for (int i = 0; i < MAXLOOP; i++)
    {
        Thread.Sleep(MAXSLEEP);
    }
}
```

Dans cet exemple (volontairement basique) de programmation linéaire les instructions vont être exécutées les unes après les autres. Cela nous permet de produire le résultat souhaité à savoir l'itération sur un nombre défini. L'instruction Thread.Sleep(MAXSLEEP) permet ici de simuler un traitement long.

Maintenant que ce passe t-il si nous souhaitons toujours effectuer ces opérations avec pour seule contrainte la vitesse d'exécution ?

Prenons, dans un premier temps, le cas d'un ordinateur n'ayant qu'un seul **processeur** (tant physique que logique). Afin d'améliorer la vitesse d'exécution vous pouvez threader l'exécution des instructions.

Les **threads** sont des contextes d'exécution permettant de **faire croire** à une exécution simultanée de plusieurs tâches. Attention je dis bien faire croire car dans le cas d'un ordinateur mono-processeur l'utilisation de thread ne change rien au temps d'exécution des instructions. Cela ne permet que d'alterner les instructions exécutées afin de donner une impression d'exécution simultanée. Dans notre cas cela ne change donc rien au temps d'exécution. Notre ordinateur n'ayant qu'un seul processeur la programmation parallèle ne peut pas être utilisée.

Passons au cas où nous possédons un ordinateur ayant au moins deux processeurs (encore une fois physique ou logique peu importe).

```
private struct CustomThreadInfo
{
    public int i;
}

private static void MainThread()
{
    Thread thread1 = new Thread(new ParameterizedThreadStart(Boucle));
    Thread thread2 = new Thread(new ParameterizedThreadStart(Boucle));

    CustomThreadInfo cti1 = new CustomThreadInfo();
    cti1.i = 0;
    CustomThreadInfo cti2 = new CustomThreadInfo();
}
```

```
cti2.i = 1;

thread1.Start(cti1);
thread2.Start(cti2);

thread1.Join();
thread2.Join();
}

private static void Boucle(object o)
{
    if (!(o is CustomThreadInfo))
        throw new Exception();

    for (int i = ((CustomThreadInfo)o).i; i < MAXLOOP; i += 2)
    {
        Thread.Sleep(MXSLEEP);
    }
}
```

Dans ce cas si nous mettons en oeuvre la méthode des threads nous devrions obtenir un gain de performance x2 ? Deux processeurs utilisés par 2 threads pour 2 blocs d'instruction = gain de performance x2 ? Et bien non pas obligatoirement. Pourquoi ? Tout simplement parce que vous avez beau avoir 2 processeurs, l'OS ne vous donnera pas forcément accès aux 2 processeurs pour faire tourner vos 2 threads. En effet si l'OS juge qu'il a besoin d'un processeur pour faire tourner un autre programme, il décidera de donner un processeur pour votre application et un processeur pour l'autre application. Le fait est que si vous faites tourner le programme de démo, il est quasiment certain que vous obtiendrez un gain de performance de x2. En effet à moins de charger fortement votre CPU, l'OS essaiera d'équilibrer les threads. Potentiellement nous sommes donc revenu au cas où nous n'avions qu'un seul processeur !

Reste maintenant le troisième cas : La programmation parallèle. Dans le cas de la programmation parallèle nous allons obliger l'OS à utiliser au maximum les deux processeurs pour exécuter le code.

```
private static void MainParallele ()
{
    Parallel.For(0, MAXLOOP, i =>
    {
        Thread.Sleep(MXSLEEP);
    });
}
```

Ce code permet d'initialiser le gestionnaire d'exécution parallèle du framework et de faire une boucle for dont les itérations seront parallèles. Au niveau purement des performances nous avons encore un gain par rapport à la technique des threads.

A l'heure de l'augmentation du nombre de processeur dans les machines (aujourd'hui n'importe quelle machine récente possède au moins 2 coeurs), le développement d'application sachant utiliser correctement l'ensemble de la puissance est un enjeu énorme. Quoi de plus frustrant que de voir votre programme "freezer" parce qu'il utilise 100% d'un processeur alors que les 7 autres sont à 10 % ?


I-C - Threads vs programmation parallèle

A ce stade de l'article une précision concernant les threads et la programmation parallèle s'impose.

En fait à la vue de la démonstration il est assez facile de penser que thread et programmation parallèle sont la même chose. Or il n'en est rien. La technique des threads permet l'exécution de deux tâches de manière isolée et pseudo-simultanée. En effet je rappelle que rien ne garanti que vos deux threads soient exécutés en même temps sur un processeur chacun.

La programmation parallèle en revanche vous apporte cette certitude. Bien qu'elle soit basée sur les threads, la programmation parallèle vous assure que vos tâches sont exécutées sur le nombre maximal de processeur de manière simultanée. Cette assurance ne peut pas être obtenue avec les threads "de base" du framework.

Outre cette considération de performance, les "parallel extensions" vous apportent également une simplicité indéniable. Reprenez les exemples de code du début et dites-moi si vous préférez débogger l'exemple avec les threads ou celui avec les parallel extensions ? De plus les parallel extensions mettent à disposition des développeurs toute une architecture de gestion des exceptions à l'intérieur des tâches parallélisées. Lorsque vous gérez des threads: Qui doit gérer les exceptions ? Le thread en lui-même ou le thread créateur ? Grâce aux parallel extensions cette problématique n'existe plus, les exceptions sont gérées dans le code appelant comme si tout s'exécutait de manière linéaire. La stratégie mise en place par les concepteurs des parallel extensions consiste à encapsuler l'exception dans une AggregateException afin de pouvoir faire la différence entre les exceptions "normales" et les exceptions survenues dans les tâches parallèles.

 *Visual Studio 2008, lors du débogage d'une application utilisant les parallel extensions, introduit un biais dans la gestion des exceptions. En effet il considère que l'exception soulevée dans une itération parallèle n'est pas gérée même si vous avez un try/catch dans le code appelant. En réalité il n'en est rien, le try/catch fonctionne parfaitement.*

Dans les exemples de code nous voyons souvent des boucles parallélisées mais il serait très dommage de réduire la programmation parallèle à cette seule facette. En effet une des grandes forces des parallel extensions est bien d'apporter une simplicité dans la mise en oeuvre de boucle parallélisée mais également d'apporter une simplicité dans la définition et l'exécution de tâche parallèle.

Attention toutefois à l'enthousiasme des premiers moments. Toutes les tâches ne peuvent pas être parallélisées et certaines ne doivent pas l'être sous peine de réduire les performances ! De manière générale les tâches qui peuvent être threadées, peuvent être parallélisées. Il existe cependant quelques exceptions. Toujours de manière générale, les tâches nécessitant de nombreux accès en écriture sur des membres d'objets ne sont pas forcément bonnes à paralléliser. En effet à partir du moment où vous souhaitez écrire des données dans, par exemple une collection, vous devez vous assurer que celle-ci est thread-safe. C'est-à-dire que vous devez être sûr que deux écritures ne puissent pas être faites au même moment sous peine de voir une des deux écritures remplacée par la suivante. Vous devez également savoir que la majorité des collections "de base" du framework ne sont pas thread-safe !. C'est pourquoi l'équipe de développement des "parallel extensions" a mis à disposition des développeurs 3 nouvelles collections génériques qui sont elles thread-safe : System.Threading.Collections.ConcurrentQueue T , System.Threading.Collections.ConcurrentStack T et System.Threading.Collections.BlockingCollection T. Toutefois l'utilisation de mécanisme d'accès exclusif est relativement coûteux en terme de performance et plus vous multipliez les accès en écriture plus vous obligez le système à poser des verrous et à les relâcher, plus vous augmentez le temps pour écrire vos données.

II - Premier programme utilisant les "parallel extensions"

Pour ce premier exemple de programmation parallèle nous allons jouer la simplicité. En effet le projet Parallel extension n'a pas fait l'impasse sur linq et nous avons donc droit à une version de linq utilisant les "parallel extensions" nommée plinq.

Dans les faits PLinq n'est rien de plus que des méthodes d'extensions permettant d'ajouter les méthodes AsParallel<T> et AsParallel aux collections de type IEnumerable.

De ce fait il devient extrêmement simple d'itérer de manière parallèle sur n'importe quelles collections et ainsi de voir les temps de traitement diminuer.

Ceci est d'autant plus vrai que la source de données est longue à répondre aux requêtes.

II-A - Premier exemple : Requête select sur une base de données

Dans ce premier exemple nous allons nous pencher sur une chose que tout le monde est amené à faire : Un select sur une base de données. Pour cela nous allons partir de la syntaxe Linq "normale" pour ensuite introduire la syntaxe PLinq. Cela suppose donc que vous avez les connaissances de bases en linq.

```
var datas = from user in dataContext.Users select user;
```

Avec ce morceau de code nous demandons à linq de nous faire un simple select sur la table Users et d'en extraire tous les users.

Vient maintenant la question du jour : Comment paralléliser cet accès aux données ?

La réponse est simple et elle est contenue dans le code suivant :

```
var datas = from user in dataContext.Users.AsParallel<Users>() select user;
```

Non non il n'y a pas une erreur de copier/coller dans l'article ;)

En effet utiliser les "parallel extensions" avec linq est vraiment très facile. Les utiliser judicieusement est déjà un peu plus dur mais ce n'est pas le sujet. Revenons sur le code. Entre le premier exemple et le deuxième il n'y a qu'un changement : AsParallelUsers. Cette méthode d'extension suffit à mettre en marche toute la mécanique pour que les requêtes linq sur la base de données soient effectuées de manière parallèle !

Je vous invite à tester le programme de démo sur votre machine, vous verrez que pour ce qui est de l'accès aux données PLinq est largement plus rapide que Linq.

Dans le programme de test il y a dans les deux cas un foreach vide comme ceci :

```
var datas = from user in dataContext.Users.AsParallel<Users>() select user;  
foreach (Users user in datas)  
{ }
```

Ceci n'est pas dû à un oubli de ma part. En effet pour que la requête linq soit exécuté il faut que les données soient parcourues. Sans cette boucle foreach aucunes données ne seraient chargées avant le Count(); De plus cette boucle est vide afin de ne pas "polluer" les résultats avec les temps que passerait le programme à attendre un affichage console par exemple.

II-B - Deuxième exemple : PLinq to Object

Ce deuxième exemple est plus un contre exemple qu'un véritable exemple. Il démontre que l'on peut utiliser PLinq sur des collections d'objets. Mais surtout il montre une limitation des "parallel extensions" dans le sens où le test n'est pas favorable à PLinq. En effet les tests montrent que la solution linéaire est plus rapide que la solution parallèle. Ceci s'explique par le faible temps de calcul de la regex dans la fonction IsCorrectData. Le temps passé à initialiser le moteur des "parallel extensions" et à créer de nouveau thread n'est pas compensé par le temps passé dans chaque itération de la boucle. Au final cela donne une version "parallel extensions" plus lente que la version linéaire !

Il faut donc bien retenir que tout ne peut pas être parallélisé et qu'à trop vouloir gagner en performance, vous risquez d'y perdre !

```
public void RunLineaire()  
{  
    Stopwatch stopWatch = Stopwatch.StartNew();  
    var result = this.datas.Where(s => IsCorrectData(s));  
    Console.WriteLine("Lineaire time : " + stopWatch.ElapsedMilliseconds);  
}  
  
public void RunParallelel()  
{  
    Stopwatch stopWatch = Stopwatch.StartNew();  
    var result = this.datas.AsParallel().Where(s => IsCorrectData(s));  
    Console.WriteLine("Parallelel time : " + stopWatch.ElapsedMilliseconds);  
}  
  
private bool IsCorrectData(string s)  
{  
    //!!!! Le temps de calcul n'est pas assez long pour que les parallel extensions soient intéressante !!!!  
    return myRegex.IsMatch(s);  
}
```

III - Conclusion

Les "parallel extensions" sont un merveilleux outil pour qui sait les utiliser. Leur conception à la fois simple et efficace permet à n'importe quel développeur ayant des notions de threading et d'accès concurrent aux données de jouer avec la puissance des machines multi-processeurs.

De plus avec PLinq c'est un vrai gain qui est possible sur les temps d'accès aux données et cela sans complication ou modification du code dans la mesure où la modification à faire est de rajouter AsParallel à la requête Linq.

Toutefois cet article n'aborde pas la notion de tâche et ne rentre pas dans les détails des possibilités des "parallel extensions" qui permettent de faire beaucoup plus que "simplement" du PLinq.

IV - Téléchargements et documentations

Source [Le programme de démo](#)

 [Télécharger les Parallel Extensions](#)

 [Architecture multi-coeurs dans les moteurs 3D](#)

 [Programmation parallèle pour le calcul scientifique](#)

 [Les architectures parallèles et leur programmation pour le calcul scientifique](#)

 [Le blog de l'équipe des parallel extensions](#)

 [La section MSDN dédié à la programmation parallèle](#)

 [Un excellent article sur PLinq](#)

V - Remerciements

Je tiens à remercier Cl@udius pour sa relecture de l'article, Laurent Dardenne pour ses remarques et conseils et toute l'équipe dotnet pour son soutien.