UFR Sciences et techniques de Franche-Comté

Projet semestriel
2ème année de Master Informatique

Performances de la programmation multi-thread

Auteur(s) :
Beuque Eric
Moutenet Cyril

Tuteur(s) :
Philippe Laurent

29 janvier 2009

Remerciements

Nous tenons à remercier Laurent Philippe qui nous a supervisé tout au long de ce projet et qui nous a aidé et conseillé afin de pouvoir réaliser ce rapport.

Table des matières

Introduction **3**

1 Définitions **4**
1.1 Architecture . . . 4
1.1.1 Multiprocesseur . . . 4
1.1.2 Multitâche . . . 4
1.1.3 Processus léger (thread) . . . 4
1.1.4 Processeurs multicoeurs . . . 4
1.1.5 Hyper-Threading . . . 5
1.1.6 HyperThreading (Autre définition) . . . 5
1.1.7 Technologie superscalaire . . . 5
1.2 Outils de programmation . . . 6
1.2.1 Threads POSIX . . . 6
1.2.2 OpenMP . . . 6
1.2.3 MPI . . . 6
1.2.4 TBB . . . 6
1.2.5 HMPP . . . 7
1.2.6 JSR-166y . . . 7
1.3 Aspect parallèle et distribué . . . 7
1.3.1 Calcul parallèle . . . 7
1.3.2 Calcul distribué . . . 7
1.3.3 Programmation concurrente . . . 7

2 Étude des performances **8**
2.1 Calcul de Fibonacci . . . 8
2.1.1 Principe de l'algorithme . . . 8
2.1.2 Comparaison des différentes bibliothèques . . . 9
2.1.3 Comparaison du niveau d'optimisation du compilateur en C . . . 11
2.1.4 Comparaison des machines virtuelle Java . . . 12
2.1.5 Comparaison finale entre Java et C . . . 13
2.2 Multiplication de matrice . . . 14
2.2.1 Principe de la multiplication de matrice . . . 14
2.2.2 Comparaison des technologies de processeurs . . . 15
2.2.3 Performance des threads . . . 18
2.2.4 Bibliothèque JSR166y . . . 21
2.3 Création de thread . . . 21
2.3.1 Principe . . . 21
2.3.2 Résultats . . . 22

3 Synchronisation **23**
3.1 Mutex . . . 23
3.2 Sémaphore . . . 23
3.3 Section critique . . . 24
3.4 Modèle producteur-consommateur . . . 24

3.5 Autres bibliothèques de synchronisation 25
3.5.1 Bibliothèque Java 25
3.5.2 Bibliothèque C Sharp 25

4 Bilan **27**
4.1 Technique 27
4.2 Humain 27

Conclusion **28**

Bibliographie **29**

A La synchronisation en CSharp **30**

Introduction

Au cours des années, les constructeurs de microprocesseurs (appelés "fondeurs"), ont mis au point un certain nombre d'améliorations permettant d'optimiser le fonctionnement du processeur. Parmi ces technologies l'hyperthreading et le multi-core.

Le but de ce sujet est donc de faire le bilan des technologies de programmation multi-thread existantes, ainsi que faire une étude des performances sur les différentes plates-formes avec différents outils de développement.

Nous avons donc été amenés à réaliser des tests sur différents systèmes d'exploitation (Linux et Windows) avec des machines utilisant des architectures spécifiques (Pentium, Pentium IV avec HyperThreading, Core2Duo, Xeon bi-processeur à quad-core...). Nous avons ainsi utilisé divers bibliothèques logicielles permettant la programmation de threads (Java, CSharp, Thread Posix, OpenMP, fork, TBB...).

Ainsi en première partie de ce rapport, nous vous présenterons un récapitulatif à travers les définitions des différentes technologies, outils et mécanismes touchant à l'univers du multi-threading. Après quoi nous vous exposerons nos résultats concernant les divers tests que nous avons réalisés, avant de vous dresser aussi un bilan des outils de synchronisation disponibles dans les différents langages, en soulignant les avantages et inconvénients qu'ils apportent.

Enfin nous conclurons ce rapport, en dressant un bilan sur ce que ce projet nous a appris sur la programmation multi-thread.

1 Définitions

Dans cette partie, vous trouverez un récapitulatif des différents termes qui touchent au monde du multi-threading. Vous y trouverez ainsi un bilan des différentes technologies des processeurs, une liste des outils de programmation multi-thread et les différents aspects de la programmation parallèle et distribuée.

1.1 Architecture

1.1.1 Multiprocesseur

Un ordinateur multiprocesseur est doté de plusieurs processeurs. Un ordinateur mono-processeur n'en comporte qu'un seul, un bi-processeur deux.

1.1.2 Multitâche

En informatique, un système d'exploitation est dit multitâche (en anglais multi-task) s'il permet d'exécuter, de façon apparemment simultanée, plusieurs programmes sur un ordinateur. On parle également de multiprogrammation. Le terme multitâche intervient au niveau logique (système d'exploitation) et est indépendant du nombre de processeurs présents physiquement dans l'ordinateur (multiprocesseur).

- Multitâche coopératif : Le multitâche coopératif est une forme simple de multitâche où chaque processus doit explicitement permettre à une autre tâche de s'exécuter. Il a été utilisé, par exemple, dans les produits Microsoft Windows jusqu'à *Windows 3.11* ou dans *Mac OS* jusqu'à *Mac OS 9*.
- Multitâche préemptif : Le traitement multitâche préemptif est une forme de traitement multitâche. Son but est de partager les temps de calcul des diverses tâches (programmes) à exécuter par le processeur d'un ordinateur.

1.1.3 Processus léger (thread)

Un processus léger [1], également appelé fil d'exécution (autres appellations connues : unité de traitement, unité d'exécution, fil d'instruction, processus allégé), est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père se partagent sa mémoire virtuelle. Par contre, tous les processus légers possèdent leur propre pile système.

1.1.4 Processeurs multicoeurs

La technologie nommée multicoeurs [2] permet d'assembler deux coeurs de processeurs côte-à-côte sur le silicium : le support (la connectique qui relie le processeur à la carte électronique) lui ne change pas. Certains éléments, comme par exemple l'antémémoire (la mémoire cache) peuvent être mis en commun. Les processeurs multi-coeurs sont cadencés

1. thread (en anglais)
2. multicoeurs : multicore en anglais

à des fréquences inférieures à celle des mono-processeurs, d'où une consommation et un dégagement de chaleur réduit. Il est possible de remplacer, en utilisant exactement les mêmes supports, deux processeurs par leur équivalent multicoeur. Ceux-ci pourront par exemple fonctionner à la fréquence de 2,8 Ghz. La machine résultante sera donc dotée de deux processeurs, mais de quatre coeurs, la puissance de calcul étant maintenant de 4 x 2,8, soit 11,2 Ghz (auxquels, il faut soustraire (deux fois) quelques pourcents). Associée à la technique de l'hyperthreading, cette technologie est en pleine essor. Sun Microsystems étudie en 2008 la construction de multicoeurs en trois dimensions, c'est-à-dire avec une zone de recouvrement des deux coeurs (donc superposés) permettant de mettre en place des canaux de communication entre eux. Ces architectures ne sont intéressantes que pour les programmes qui se prêtent simultanément aux deux techniques précitées (multicoeurs et hyperthreading).

1.1.5 Hyper-Threading

HT (en français Hyper-Flots ou Hyper-Flux) est le nom donné par *Intel* à son adaptation du *SMT* (Simultaneous Multi Threading) à deux voies dans le *Pentium IV*. Le premier modèle grand public de la gamme à en bénéficier fut le 3,06 Ghz basé sur le core Northwood. Le processeur *Xeon Northwood* en bénéficia auparavant. Schématiquement, l'hyper-threading consiste à créer deux processeurs logiques sur une seule puce, chacun doté de ses propres registres de données et de contrôle, et d'un contrôleur d'interruptions particulier. Ces deux unités partagent les éléments du coeur de processeur, le cache et le bus système. Ainsi, deux sous-processus peuvent être traités simultanément par le même processeur. Cette technique multitâche permet d'utiliser au mieux les ressources du processeur en garantissant que des données lui sont envoyées en masse. Elle permet aussi d'améliorer les performances en cas de cache miss.

1.1.6 HyperThreading (Autre définition)

La technologie *HyperThreading* consiste à définir deux processeurs logiques au sein d'un processeur physique. Ainsi, le système reconnaît deux processeurs physiques et se comporte en système multitâche en envoyant deux threads simultanés, on parle alors de *SMT*[3]. Cette "supercherie" permet d'utiliser au mieux les ressources du processeur en garantissant que des données lui sont envoyées en masse. [1]

1.1.7 Technologie superscalaire

La technologie superscalaire (en anglais superscaling) consiste à disposer plusieurs unités de traitement en parallèle afin de pouvoir traiter plusieurs instructions par cycle.[1]

3. Simultaneous Multi Threading
1. www.commentcamarche.net

1.2 Outils de programmation

1.2.1 Threads POSIX

La bibliothèque pthread utilisée correspond à une implantation de la norme *POSIX1003.1c*. Elle fournit des primitives pour créer des activités (ou processus légers) et les synchroniser. Ces primitives sont similaires à celles fournies par diverses autres bibliothèques (Solaris LWP, Windows) ou langages (Modula-3, Java). Dans le domaine des processus légers, la norme *POSIX* s'est imposée, tout au moins dans le monde Unix.

Une documentation plus complète de la librairie pthreads est disponible dans le *man*[4].

1.2.2 OpenMP

Open Multi-Processing est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est supportée sur de nombreuses plate-formes, incluant Unix et Windows, pour les langages de programmation C/C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement. OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel. La programmation parallèle hybride peut être réalisée par exemple en utilisant à la fois OpenMP et MPI.

1.2.3 MPI

The Message Passing Interface, conçu en 1993-94, est une norme définissant une bibliothèque de fonctions, utilisable avec les *langages C* et *Fortran*. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages. Il est devenu de facto un standard de communication pour des noeuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. *MPI* a été écrit pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Il est grandement disponible sur de très nombreux matériels et systèmes d'exploitation. Ainsi, *MPI* possède l'avantage par rapport au plus vieilles bibliothèques de passage de messages d'être grandement portable (car *MPI* a été implanté sur presque toutes les architectures de mémoires) et rapide (car chaque implantation a été optimisée pour le matériel sur lequel il s'exécute). Depuis 1997, une nouvelle version de MPI est disponible : *MPI-2*.

1.2.4 TBB

Intel® Threading Building Blocks (TBB) est une bibliothèque C++ destinée à la réalisation d'application de calcul parallèle et qui permet de s'abstraire de la partie bas niveau de la gestion des threads sur des architectures multi-coeurs dans le but d'obtenir des performances optimales. Elle est basée sur l'utilisation de templates C++ et sur un style de programmation qui élimine l'implémentation propre des threads.

4. man libpthread ou man pthread_xxx

1.2.5 HMPP

Hybrid Multicore Parallel Programming est un ensemble d'outils de développement au service de la programmation multi-coeurs hybride. *HMPP* est un produit commercial de *CAPS entreprise.*

1.2.6 JSR-166y

JSR-166y est une bibliothèque qui est actuellement en développement dans le but d'être intégrée à la version de 7 de Java. Celle-ci permet de facilité la mise en place de processus parallèles avec exploitation des performances optimales. Celle-ci est très adaptée aux processeurs avec plusieurs cœurs.

On grâce à cette API, il est possible d'initialiser une liste avec une autre liste en parallèle. Un certain nombre d'exemple sont disponibles sur le site internet dédié à JSR-166 http ://artisans-serverintellect-com.si-eioswww6.com/default.asp ?W32.

Cette bibliothèque reprend un peux le fonctionnement du package concurrent de java. Mais avec des performances légèrement plus avantageuses.

1.3 Aspect parallèle et distribué

1.3.1 Calcul parallèle

En informatique, le calcul parallèle consiste en l'exécution d'un traitement pouvant être partitionné en tâches élémentaires adaptées afin de pouvoir être réparties entre plusieurs (cores de) processeur(s) opérant simultanément en vue de traiter plus rapidement que dans une exécution séquentielle des problèmes plus grands (à technologie constante).

1.3.2 Calcul distribué

Le calcul distribué ou réparti, aussi connu sous le nom de système distribué, consiste à répartir un calcul ou un système sur plusieurs ordinateurs distincts. Le calcul distribué est une forme de calcul parallèle, par contre il diffère des grappes de serveurs (cluster computing), en ce que les ordinateurs qui réalisent le calcul ne sont pas typiquement dédiés au calcul distribué et ne possèdent pas toujours les mêmes matériels et systèmes de fichiers, alors que les grappes (clusters) comprennent la plupart du temps du matériel spécifique et dédié à une tâche précise.

1.3.3 Programmation concurrente

La programmation concurrente est un style de programmation tenant compte, dans un programme, de l'existence de plusieurs piles sémantiques. Ces piles peuvent être appelées threads, processus ou tâches. Ils sont matérialisés en machine par une pile d'exécution et un ensemble de données privées. Les threads disposent d'une zone de mémoire partagée alors que les processus sont strictement isolés. La concurrence est indispensable lorsque l'on souhaite écrire des programmes interagissant avec le monde réel (qui est concurrent) ou tirant parti de multiples unités centrales (couplées, comme dans un système multiprocesseurs, ou distribuées, éventuellement en grille ou en grappe).

2 Étude des performances

Pour réaliser une étude comparative entre les différentes plates-formes, nous avons commencé par chercher des exemples d'application (un Benchmark) permettant de réaliser ces tests de performance.
Nous avons alors trouvé deux exemples intéressants :

1. Le calcul de Fibonacci : Il s'agit d'un calcul simple récursif
2. le calcul d'un produit de matrice : Le calcul de la multiplication de matrice a été modifié afin de gérer au mieux le calcul parallèle

Les benchmarks ont alors été réalisés afin de voir les performances lorsque l'application était plus ou moins parallélisée :
- sans découpage
- avec un découpage
- avec 2 découpages
- avec 4 découpages

Ceci afin de vérifier les performances sur différents types de processeurs :
- Standard : processeurs normaux sans cœurs supplémentaires
- Multi-cœurs (multicore en anglais) : processeur à plusieurs cœurs physiques
- Avec Hyperthreading

Nous avons pour cela effectué plusieurs implémentations de ces deux algorithmes dans différents langages en utilisant les bibliothèques disponibles avec ceux-ci :
- c_nothread : Implémentation en langage C sans threads
- c_pthread : Implémentation en langage C avec la bibliothèque de threads POSIX
- c_openmp : Implémentation en langage C avec la bibliothèque OpenMP
- cpp_tbb : Implémentation en langage C++ avec la bibliothèque TBB d'IBM
- CSharpNoThread : Implémentation en langage C# sans thread
- CSharpThread : Implémentation en langage C# avec threads
- JavaNoThread : Implémentation en langage Java sans thread
- JavaThread : Implémentation en langage Java avec threads
- JavaNoThreadRMI : Implémentation en langage Java sans thread avec Java RMI
- JavaThreadRMI : Implémentation en langage Java avec threads avec Java RMI

2.1 Calcul de Fibonacci

2.1.1 Principe de l'algorithme

Pour la première série de nos tests, nous avons décidé de paralléliser une fonction calculant le nombre de Fibonacci d'un entier avec un algorithme récursif (cf. algorithme 1 p9).

Algorithm 1 Fonction qui calcule le nombre de Fibonacci en récursif

Require: Un entier $n \geq 0$.

```
1: function FIBO(n)
2:     if n ≤ 1 then return n
3:     else return fibo(n − 1) + fibo(n − 2)
4:     end if
5: end function
```

Le programme prend en paramètre le nombre à calculer et le nombre de fois que l'on doit le calculer (cf. algorithme 2) :

Algorithm 2 Calcule n fois le nombre de Fibonacci d'un entier

```
1: nFibo ← lire()                          ▷ Lecture du nombre à calculer
2: nbFois ← lire()                         ▷ Lecture du nombre de fois à calculer
3: dateDebut ← now()
4: for i = 1 to nbFois do                  ▷ Début de la région parallèle
5:     fibo(nFibo)
6: end for                                 ▷ Fin de la région parallèle
7: dateFin ← now()
8: duree ← dateFin − dateDebut
9: ecrire(duree)
```

Pour chaque implémentation que l'on fera, on parallélisera la boucle (cf. algorithme 2), afin que chaque thread calcule donc le nombre de Fibonacci une fois. On crée donc autant de thread que l'on doit calculer ce nombre.

2.1.2 Comparaison des différentes bibliothèques

Nous avons d'abord cherché à comparer les performances de chaque librairie lorsque l'algorithme est ou n'est pas parallélisé. Notons que les deux tests ci-dessous ont été effectués sous Linux avec la machine virtuelle OpenJDK1.6 pour les tests en Java, et avec Mono 1.9.1 pour les tests en CSharp, et les programmes en C ont été compilés avec GCC 4.3.2.

Test avec un thread : Voici tout d'abord les résultats du programme sans aucun thread. Ici on calcule donc une seule fois le nombre de Fibonacci, soit fibo(n) (cf. algorithme 1 p9).

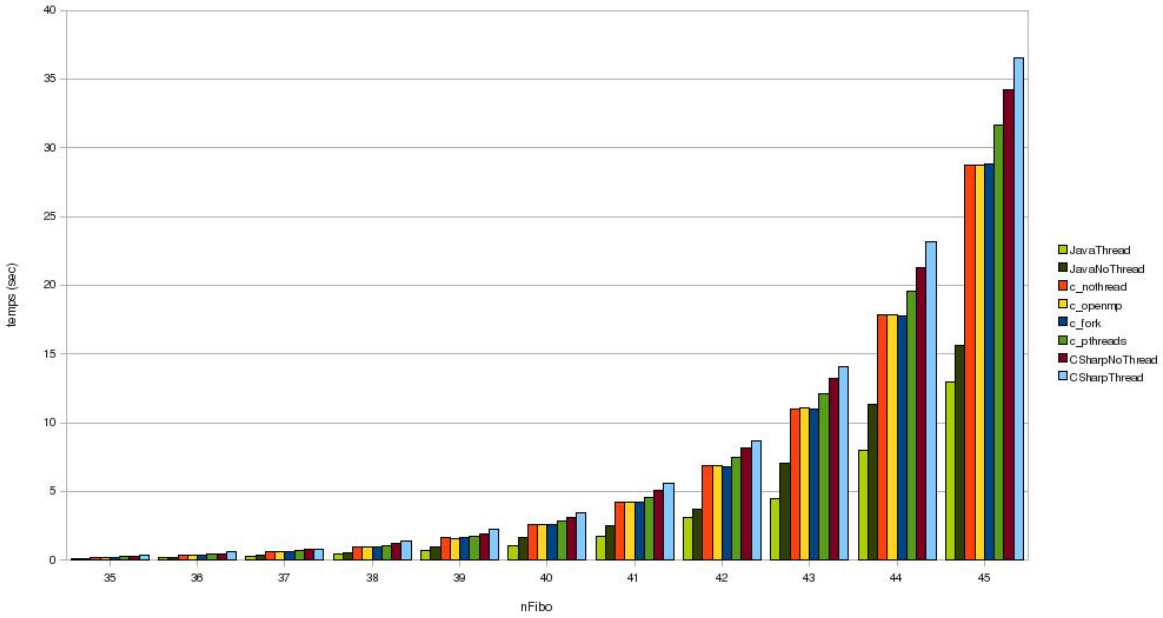

Figure 1 – *Calcul de 1 fois le nombre de Fibonacci sur un Linux Core2Duo T9300 2,5GHz*

On constate alors (figure 1) que sans trop de surprise la plate-forme Mono est dépassée en performance par les autres technologies. Par ailleurs et logiquement, étant donné que l'on n'utilise pas leur potentiel, les implémentations prenant en compte les threads sont légèrement affectés de ce fait, et sont donc ralenties, sauf dans le cas du Java. En revanche, la chose surprenante est que le Java avec threads ou non semble beaucoup plus rapide (pratiquement deux fois) que toutes les autres solutions en C.

Test avec deux threads : Et voici maintenant le résultat avec deux threads, ici on calcule donc deux fois le nombre de Fibonacci, soit 2*fibo(n) (cf. algorithme 1 p9).

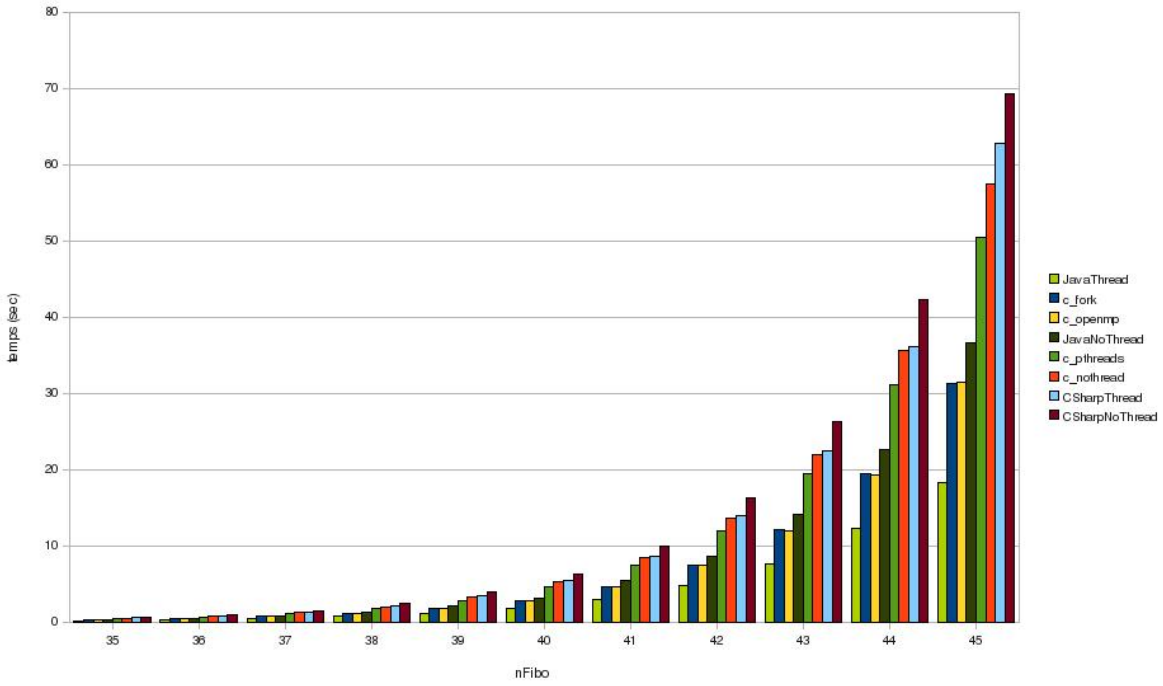

Figure 2 – *Calcul de 2 fois le nombre de Fibonacci sur un Linux Core2Duo T9300 2,5GHz*

On s'aperçoit alors qu'il y a bien du changement, et comme on pouvait s'en douter, les implémentations avec threads deviennent bien plus rapides que celles sans les threads (cf. figure 2). En effet, on n'est pas loin d'être deux fois plus rapide dans certains cas. On pourrait donc en déduire que le *Dual Core* effectue bien sa tâche et réparti bien les deux threads sur chaque cœur. En revanche dans le cas du *C#*, le gain à l'utilisation de thread semble relativement faible, ce qui confirme le manque de performance de Mono, qui reste encore derrière tout le reste. A part ça, on constate toujours que Java semble plus performant que le *C*, mais on retiendra quand même que les trois meilleures solutions semblent le Java avec thread, les *fork* et *OpenMP*. Les threads *POSIX* quand à eux montrent qu'ils sont relativement moins performants que ces derniers.

2.1.3 Comparaison du niveau d'optimisation du compilateur en C

Les constats effectués plus haut nous ont amené à essayer de comprendre pourquoi le *Java* était plus rapide que le *C*. Nous avons alors pensé à essayer de voir si le niveau d'optimisation du compilateur pouvait avoir un impact important sur les performances du programme. Nous avons donc effectué les tests en faisant varier ce niveau à savoir, celui par défaut contre celui en *-O1* (premier niveau d'optimisation) et *-03* (troisième niveau). Ces statistiques représentent une moyenne entre le calcul avec un thread et avec deux threads sur le nombre de Fibonacci. Voici donc les résultats que nous avons obtenus :

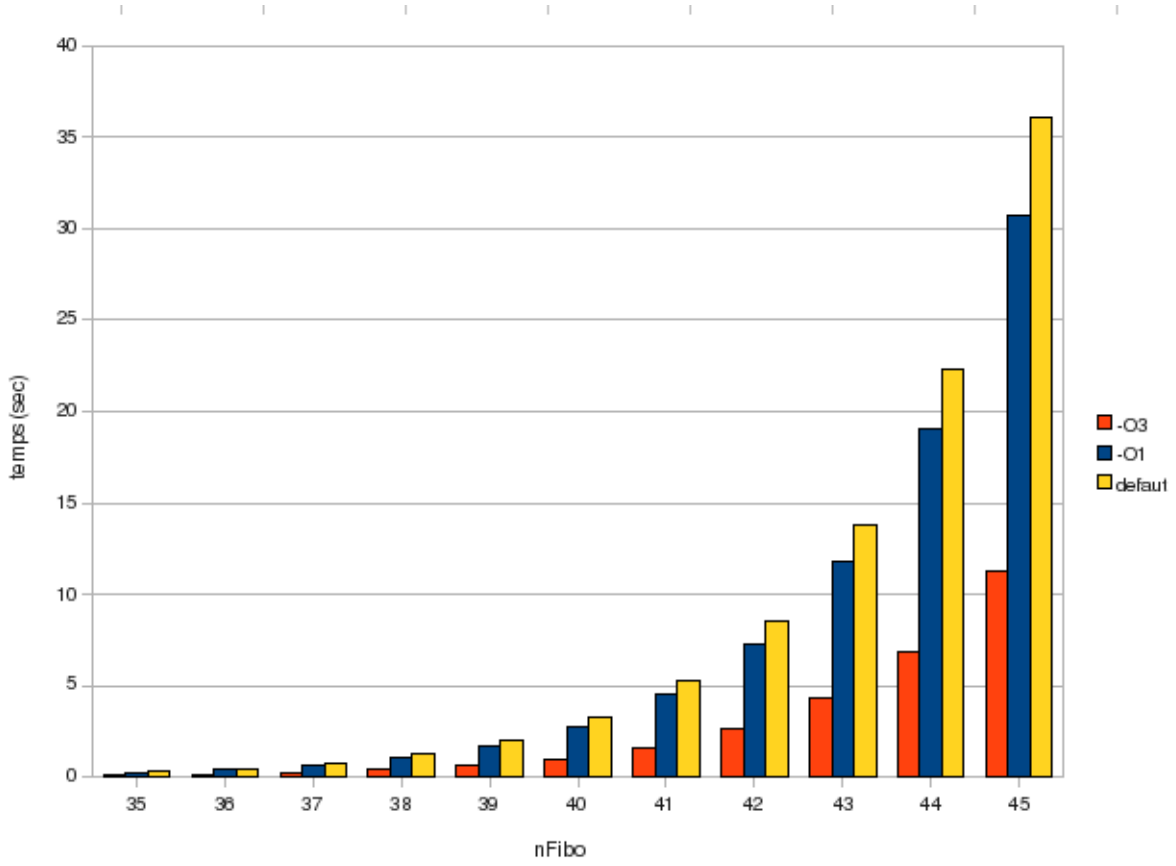

Figure 3 – *Comparaison du niveau de compilation de GCC avec Fibonacci sur un Linux Core2Duo T9300 2,5GHz*

On se rend compte alors (figure 3) de l'impact, qu'implique cette option. Sans conteste, plus on augmente le niveau de compilation, plus les performances sont accrues. En effet, le temps est carrément divisé par trois entre le niveau de compilation par défaut et le niveau 3.

2.1.4 Comparaison des machines virtuelle Java

Étant donné les résultats obtenus avec le niveau de compilation en C, nous nous somme demandés si les performances entre les machines virtuelles Java (JVM) avec un impact aussi conséquent. Nous avons donc comparés *OpenJDK1.6*, contre les *SunJDK1.5* et *SunJDK1.6*. Ces statistiques représentent une moyenne entre le calcul avec un et deux threads sur le nombre de Fibonacci. Voici donc ce qu'il en ressort (figure 4) :

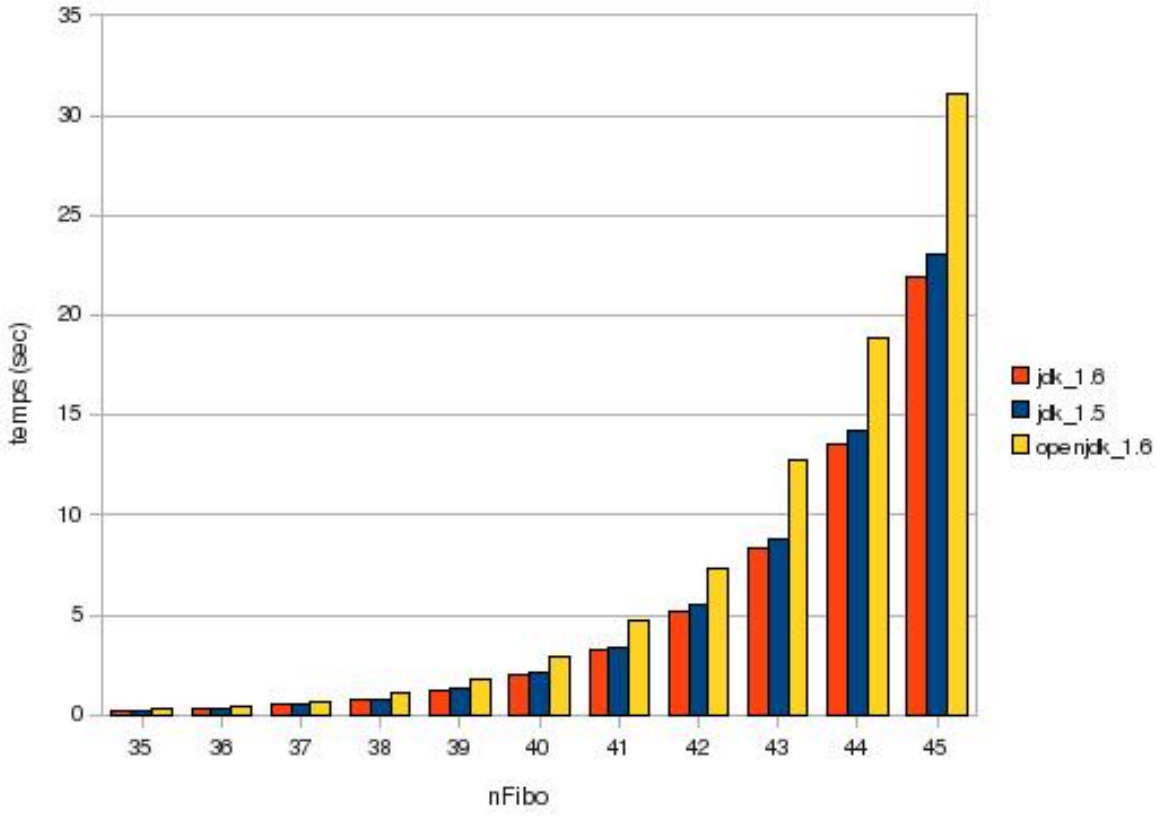

Figure 4 – *Comparaison des JVM avec Fibonacci sur un Linux Core2Duo T9300 2,5GHz*

Les conséquences sont relativement moins importantes, la *JDK1.6* de Sun quasiment égale à sa *JDK1.5*, en revanche la *OpenJDK* est légèrement derrière.

2.1.5 Comparaison finale entre Java et C

Finalement, étant donné les résultats de comparaison entre les *JVM* et le niveau de compilation de *GCC*, on en déduit que les résultats des premiers tests n'étaient pas totalement justes. Il fallait donc refaire le test entre Java et C en prenant le meilleur de chaque, c'est à dire la *JVM SunJDK1.6* et le niveau 3 de compilation de *GCC*. Ces statistiques représentent une moyenne entre le calcul avec un et deux threads sur le nombre de Fibonacci.

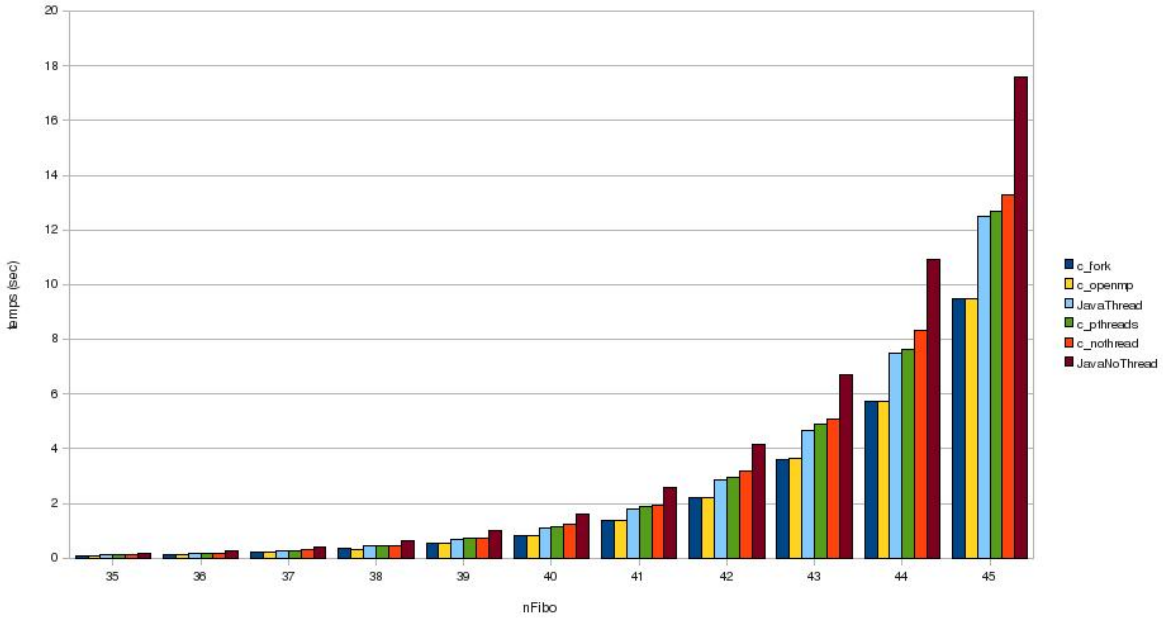

Figure 5 – *Comparaison entre SunJDK1.6 et C en -O3 avec Fibonacci sur un Linux Core2Duo T9300 2,5GHz*

Et là, revirement de situation, on s'aperçoit alors (figure 5) que le C repasse donc bien devant le Java, ce qui paraissait à la base plus plausible. On retiendra alors que *OpenMP* et les *forks* sont les solutions finales offrant les meilleurs performances. Ainsi on peut tout de même souligner de le JAVA tient plutôt bien la route, compte tenu des avantages qu'il apporte en matière de portabilité et autres...

2.2 Multiplication de matrice

Ce test doit permettre de mettre en lumière l'aspect des performances d'un calcul parallèle sur une mémoire commune contrairement au calcul de Fibonacci qui ne nécessite aucun partage de données.

2.2.1 Principe de la multiplication de matrice

Pour calculer le produit de matrice, il suffit de lire une ligne et une colonne dans les deux matrices à multiplier. Le résultat de la multiplication est stocké dans la matrice résultat. Ce calcul peut donc être parallélisé simplement sans risquer de mettre en avant des problèmes tels que la concurrence d'écriture et de lecture. Comme pour le calcul de la matrice inverse.

Le calcul a été réalisé afin de pouvoir être exécuté de manière distante en Java RMI. Ainsi le calcul s'effectue de la manière suivante. A chaque calcul deux lignes correspondant à une ligne de la matrice A et une colonne de la matrice B. Le résultat étant écrit dans la matrice résultat.

On notera que l'algorithme consiste en fait à l'utilisation d'une imbrication de trois boucles *for*, dont seul les deux dernières font parties de la région parallélisée.

Algorithm 3 Calcul la multiplication d'une matrice A par B, et met le résultat dans C

```
1: mSize ← lire()                                   ▷ Lecture de la matrice
2: dateDebut ← now()
3: A ← initialiser(mSize)        ▷ Initialise et remplie la matrice en fonction de sa taille
4: B ← initialiser(mSize)
5: for i = 0 to mSize do                            ▷ Début de la région parallèle
6:     for j = 0 to mSize do
7:         for k = 0 to mSize do
8:             C[i][j] ← C[i][j] + A[i][k] * B[k][j]
9:         end for
10:    end for
11: end for                                         ▷ Fin de la région parallèle
12: dateFin ← now()
13: duree ← dateFin − dateDebut
14: ecrire(duree)
```

2.2.2 Comparaison des technologies de processeurs

Nous avons voulu profiter des tests utilisant le calcul de matrice pour comparer plusieurs types de machine. En effet, nous avons voulu voir si le gain de performance annoncé par la technologie HyperThreading, était comparable à celle réalisée par l'utilisation des machines à double coeurs. Nous nous sommes basés pour cela, sur des tests utilisant deux threads pour le calcul.

Intel Core2Duo T9300 2,50GHz :

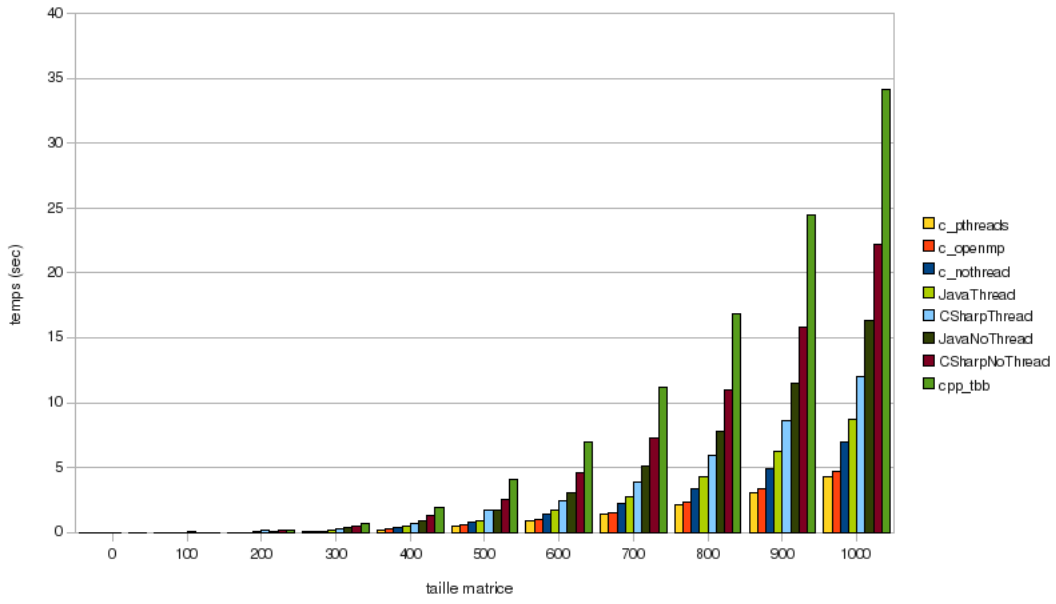

Figure 6 – *Multiplication d'une matrice avec 2 threads sur un Linux Core2Duo T9300 2,5GHz*

On constate alors que l'on obtient logiquement les mêmes résultats que ceux du calcul de Fibonnacci sur ce type de machine à savoir que l'utilisation de threads augmente sensiblement les performances. A noter tout de même que les threads *POSIX* semblent avoir des performances équivalentes voir mieux que celles de *OpenMP* sur ce calcul. Cependant, ces deux librairies devancent pareillement le C sans thread, le *JAVA* et le *CSharp avec Mono*. A noter que pour ce test, nous avons testé l'API *TBB*, cependant on constate que les performances sont étonnamment faibles par rapport à ce qu'on aurait pu attendre (moins bonne que le *CSharp avec Mono*). On notera que comme le précise la documentation [5], TBB n'est pas fait pour remplacer des librairies comme OpenMP. Cette API est plutôt conçue pour réaliser des choses plus complexes que OpenMP ne permet pas de réaliser de façon simple. C'est pourquoi, on peut déduire que TBB n'est pas censé fournir des performances meilleures que les autres API.

Pentium 4 avec HyperThreading :

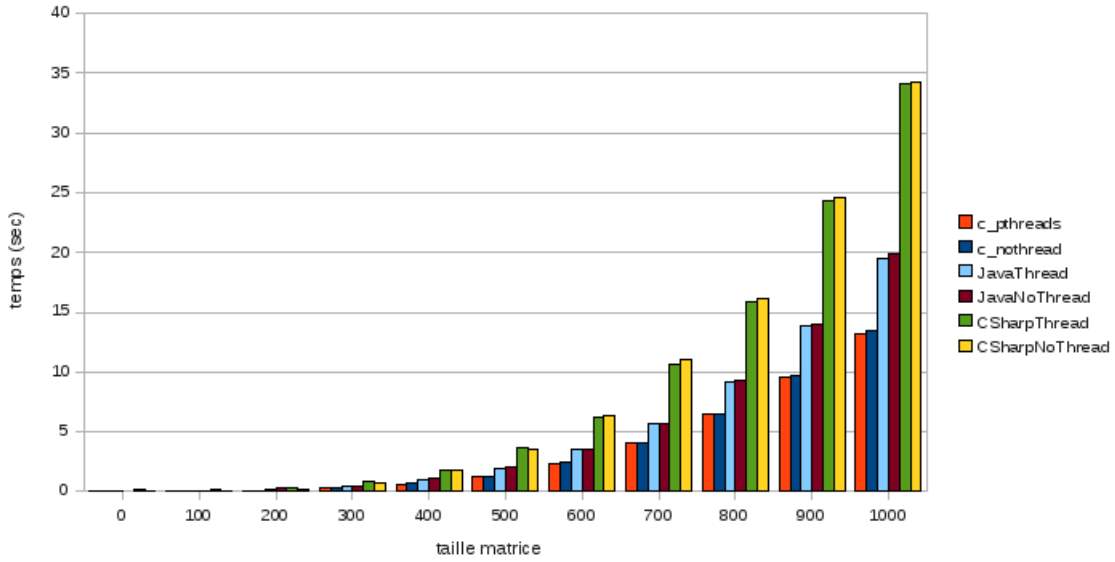

Figure 7 – *Multiplication d'une matrice avec 2 threads sur un Linux P4 HyperThreading*

Ce test nous a permis de nous rendre compte que l'utilisation de threads sur une machine avec HyperThreading offre un gain de performances très minimes voir nul, par rapport au même programme n'utilisant pas de thread. Malheureusement, pour des raisons techniques, nous n'avons pas pu tester *OpenMP* sur cette plate-forme. Toutefois, on peut en déduire que cette technologie n'offre aucun intérêt dans ce cas de figure.

Pentium Xeon avec 2 processeurs de 4 coeurs chacun :

5. Doc TBB : http ://www.threadingbuildingblocks.org/wiki/index.php ?title=Using_TBB

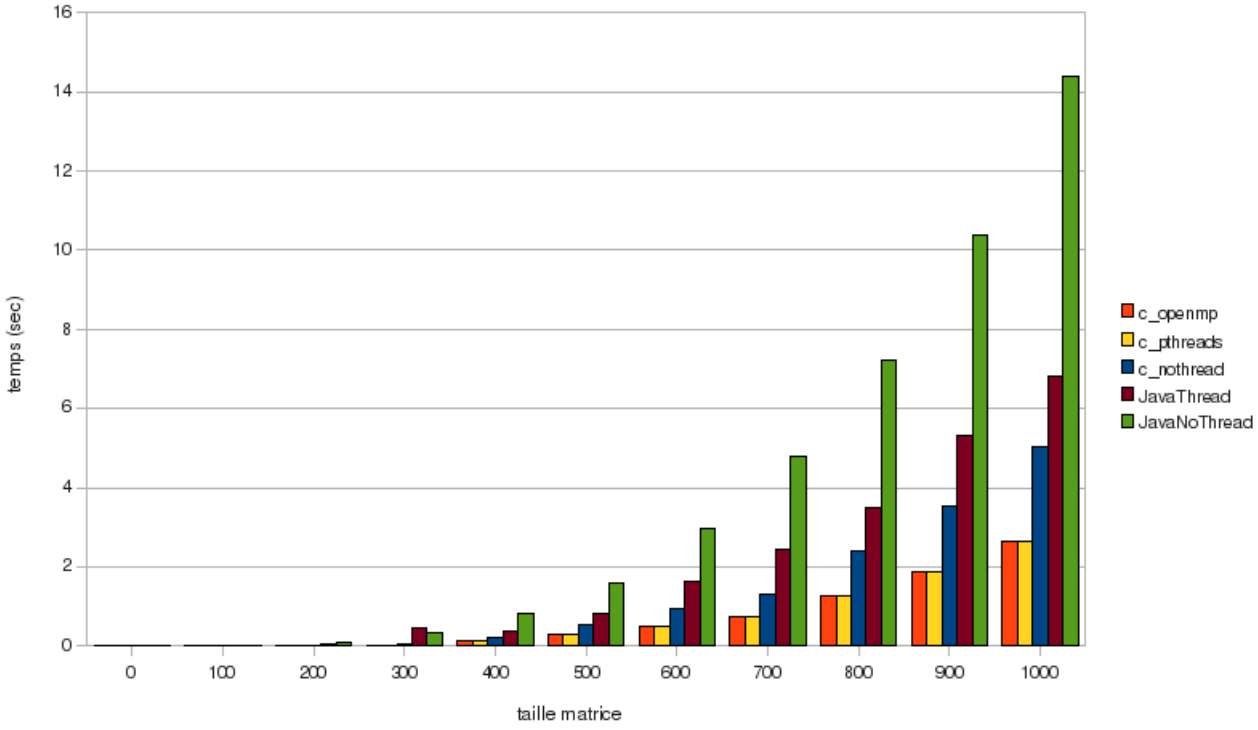

Figure 8 – *Multiplication d'une matrice avec 2 threads sur un Linux Pentium Xeon X5460 3.16GHz (2 processeurs à 4 coeurs)*

On constate ici que dans le cas où l'on utilise seulement deux threads sur une architecture de ce type les résultats sont identiques à l'utilisation d'un Core2Duo à savoir qu'un programme avec thread est deux fois plus rapide que sans thread.
Maintenant voyons ce qu'il se passe lorsqu'on augmente le nombre de thread :

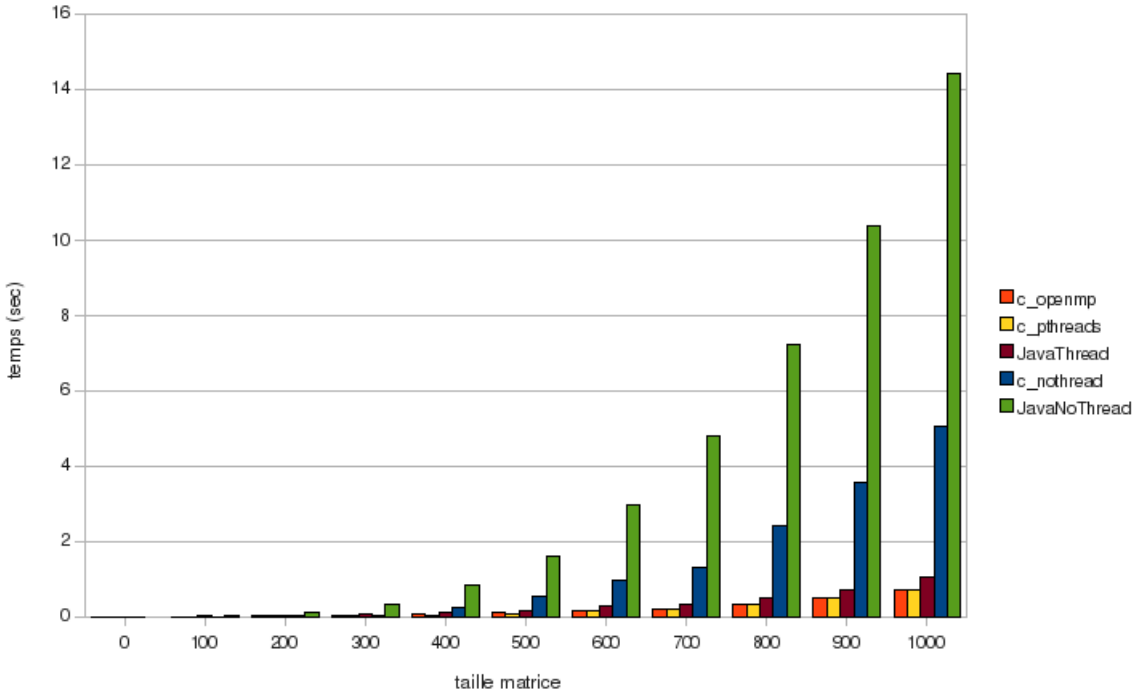

Figure 9 – *Multiplication d'une matrice avec 8 threads sur un Linux Pentium Xeon X5460 3.16GHz (2 processeurs à 4 coeurs)*

On constate alors que le gain de performances est considérable. On est par exemple pas loin de 15 fois plus rapide entre le Java sans thread et le Java avec thread.

2.2.3 Performance des threads

Nous avons réalisé les tests suivants afin de mettre en évidence le gain apporté à un programme réalisé avec thread par rapport à un programme réalisé sans thread.

D'autre part les tests ont été réalisés sur différentes plates-formes (XP et linux) et sur différents types de processeurs. Les ordinateurs possèdent des caractéristiques différentes ce n'est donc pas les performances du test mais le gain d'exécution entre une version sans thread et avec 1, 2 ou 4 threads. Des tests supplémentaires ont été effectués avec plus de 8 threads afin de constater jusqu'où le gain (si gain il y avait) pouvait aller.

Types de caractéristiques différentes :

- Fréquence de processeur
- Fréquence d'horloge
- HyperThreading, Dual core

Processeur T5500

Ce type de processeur est un processeur double coeurs cadencé à 1.66GHz. Comme on peut le voir sur la figure 11 il n'y pas de gain en performance lorsque l'on exécute le calcul avec quatre threads. Ce processeur optimise bien l'utilisation des ressources du processeur, avec un thread par coeur.

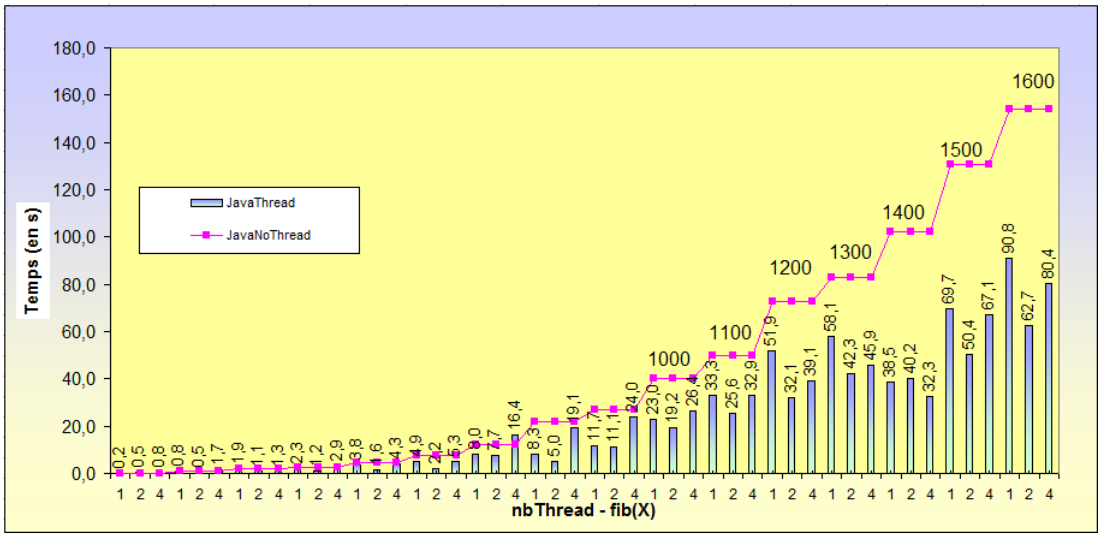

Figure 10 – *Multiplication d'une matrice sur un T5500 1.66Ghz sous Windows*

Processeur P4 D

Comme le montre la figure 11 (p.19) la gestion des threads est mieux gérée sur un *Pentium 4 D*[6]. L'amélioration observée vient du gain observé lorsque l'on utilise quatre threads pour multiplier deux matrices.

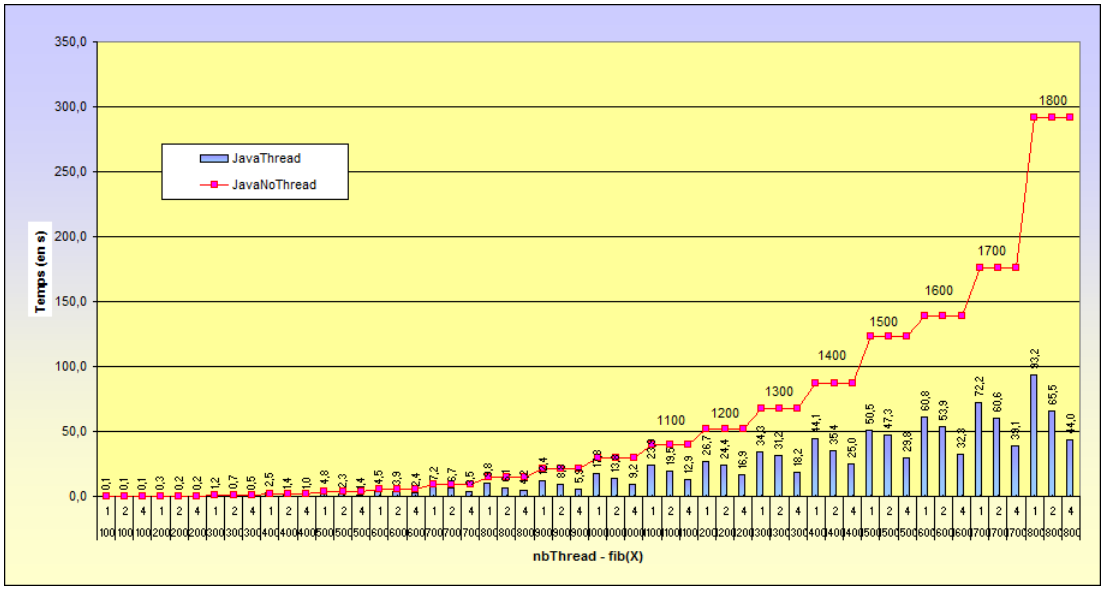

Figure 11 – *Multiplication d'une matrice sur un P4 D 2.8GHz sous Linux*

6. D pour dual core

Comme on peut le remarquer, ce type de processeur connaît une amélioration lorsque le nombre de threads croît. Ce qui n'est pas me cas avec le processeur T5500 précédemment observé.

Pentium Xeon Quad core X 2

Le même test a été effectué cette fois si sur une machine dotée de performances dédiées au travail multitâche. L'algorithme est le même que celui utilisé lors des précédant tests. Cette fois si le nombre de thread maximum testé est supérieur puisque la machine compte pas moins de 8 cœurs à son actif.

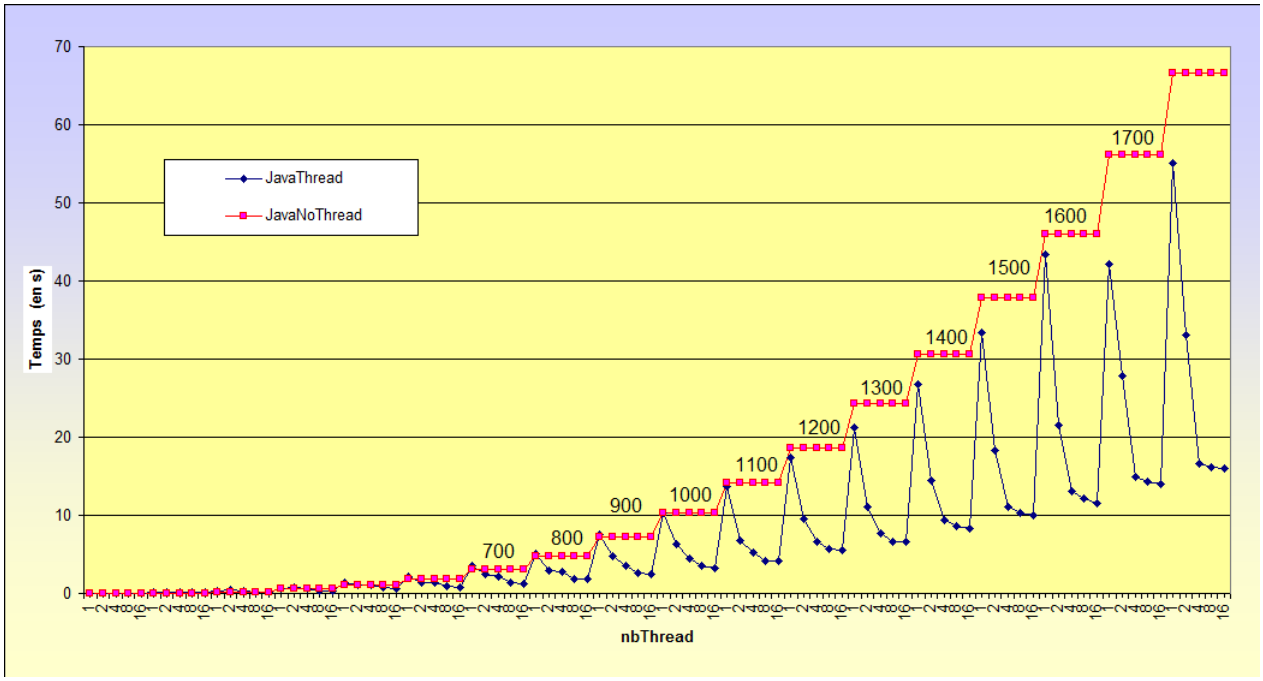

Figure 12 – *Multiplication d'une matrice sur un Pentium Xeon à 8 cœurs sous Linux*

Sur la figure 6, les tests ont été réalisés en augmentant de 100 en 100 la taille de la matrice jusqu'à arriver au calcul d'une matrice de 1800x1800. Les essais ont été effectués successivement sans thread en rose, avec 1, 2, 4, 8 et 16 threads en bleu.

Comme on peut le constater, les performances sont 2,5 fois plus rapide que le Pentium 4M (fig. 11). On aussi voir que les performances commencent à être intéressantes à partir de 4 threads. Au delà, il n'y pas de grosses améliorations. Cette constatation est plutôt surprenante, en effet on aurait pu penser que d'avantage de cœur aurait du permettre d'avantage de travail en parallèle.

Difficile juste avec ces tests de définir clairement ce qui peut bien se passer au niveau de la mémoire partagée entre les 2 Quad Core. Mais il semble que cela ne soit pas très performant.

2.2.4 Bibliothèque JSR166y

Le test suivant a été effectué à partir d'un exemple récupéré sur le site internet de «jsr166y overview» [7]. Celui-ci met à disposition la bibliothèque jsr166y.jar et un certain nombre d'exemple d'utilisation de celle-ci. Entre autre un test de multiplication de matrice.

Cet exemple n'a qu'un seul inconvénient, c'est que le nombre de thread, la taille de la matrice et le grain doivent être des puissances de 2. Ce test intervient à la fin du projet, c'est ainsi que nous n'avons pas effectué plus de test ni regardé plus en détails le contenu des sources. Les tests ont été effectués sur le double quad Core sur des matrices de taille en puissance de 2.

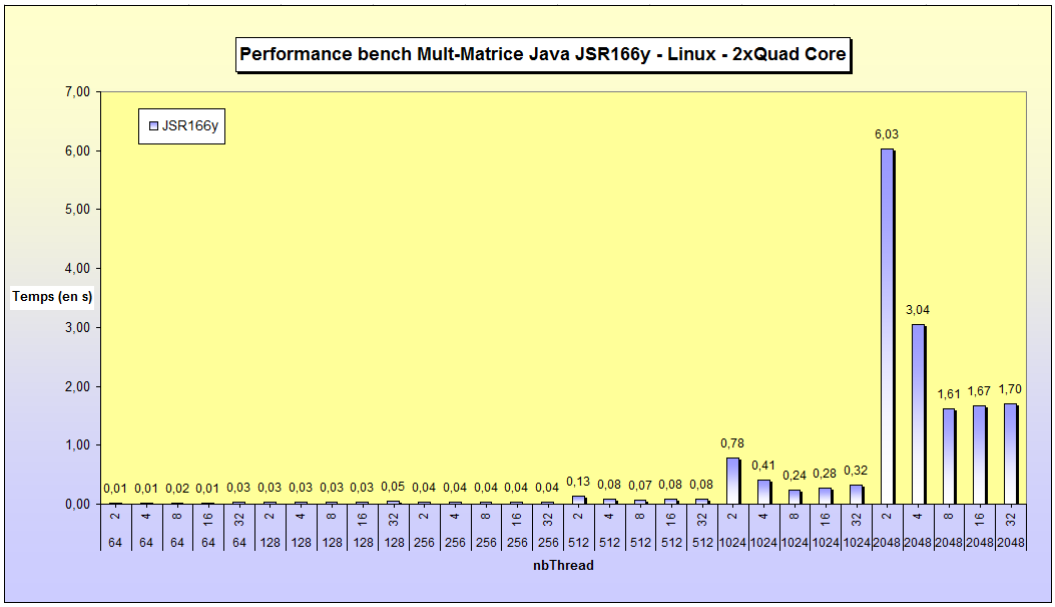

Figure 13 – *Multiplication d'une matrice sur un Pentium Xeon à 8 cœurs sous Linux*

Le résultat des tests est très surprenant, il est assez facile de conclure que ce programme fonctionne plus vite que celui que nous avons testé jusque là. Que dire sur le comment du pourquoi.

2.3 Création de thread

Comme il l'a été observé dans les tests précédant avec Fibonacci (p.11) les threads Java sont relativement performants. Afin de vérifier que les performances ne sont pas obtenues au détriment de la mémoire utilisée, nous avons réalisé des tests sur la mémoire utilisé lors de la création de plusieurs threads.

2.3.1 Principe

Pour réaliser ce test on crée 1000 threads. La création d'un thread est suivie de la destruction de celui-ci. A la fin du test le programme est mis en pause. Ensuite l'utilisation

7. jsr166y overview : http ://artisans-serverintellect-com.si-eioswww6.com/default.asp ?W9

de la mémoire est calculée grâce à l'outil TPTP d'éclipse. Celui-ci permettant de donner avec précision la liste des objets en mémoire.

Algorithm 4 Créer 1000 thread puis attendre

```
1: for i = 1 à 1000 do
2:     Créer thread                          ▷ Créer thread
3:     Attendre fin thread                   ▷ Attendre fin thread
4: end for
```

2.3.2 Résultats

Estimations du temps de création d'un thread : Nous avons donc pu estimer le temps de création d'un thread pour chacune des librairies (figure 14). On constate alors, que là où en règle générale *OpenMP* est le meilleur en *C*, c'est pour une fois les threads *POSIX* qui proposent les meilleures performances. Il devance de loin le temps de création d'un thread en *JAVA* qui prend 10 fois plus de temps, et de plus de 60 fois celui en *CSharp avec Mono*.

c_openmp	c_pthreads	CSharpThread	JavaThread
21.4 µs	14.6 µs	8905.1464 µs	161.4 µs

Figure 14 – *Temps de création d'un thread sur un Linux Core2Duo T9300*

Utilisation de la mémoire par les threads : Nous avons utilisé le plugin TPTP dans Eclipse pour calculer l'occupation mémoire de plus de 10000 threads en JAVA. Le résultat attendu ne fut pas très pertinent, en effet à la fin de l'exécution des 10000 threads la mémoire restaient occupée par 10000 bytes. Il n'y a aucune garantie que la création de ces bytes soit due à la création des threads, on peut juste remarquer que la libération mémoire n'était pas opérée.

3 Synchronisation

La synchronisation consiste à coordonner l'accès en lecture et en écriture sur une ou plusieurs ressources entre les différents processus d'un programme. Pour cela, plusieurs techniques peuvent être employées. Nous verrons ainsi dans cette partie les différentes méthodes qui peuvent être utilisées, ainsi que leur implémentation dans les divers langages. Elle est un aspect important dans la programmation multi-thread car elle permet de coordonner l'accès en lecture et écriture sur les données partagées par les processus d'un programme.

3.1 Mutex

Un Mutex (anglais : Mutual exclusion, Exclusion mutuelle) est une primitive de synchronisation utilisée pour éviter que des ressources partagées d'un système ne soient utilisées en même temps. Les mutex possèdent donc généralement deux fonctions : une pour verrouiller le mutex et une autre pour le déverrouiller.

En C avec les threads POSIX : Les mutexs sont inclus dans la librairie gérant les threads POSIX et les fonctions sont définies dans le fichier *pthread.h*. On crée alors un objet *pthread_mutex_t* que l'on initialise avec la fonction *pthread_mutex_init()*. On a alors la fonction *pthread_mutex_lock()* pour verrouiller le mutex et *pthread_mutex_unlock()* pour le déverrouiller. Une fois l'utilisation du mutex terminer ne pas oublier de le détruire avec *pthread_mutex_destroy()*.

En C avec OpenMP : Il existe des fonctions pour la gestion de mutex fournies avec OpenMP. Le fonctionnement est identique à celui des threads POSIX. On crée un objet *omp_lock_t* que l'on initialise avec la fonction *omp_init_lock()*. On a alors la fonction *omp_set_lock()* pour verrouiller le mutex et *omp_unset_lock()* pour le déverrouiller. Enfin on peut le détruire avec la fonction *omp_destroy_lock()*.

En Java : Il faut utiliser la classe *ReentrantLock* présente dans *java.util.concurrent.lock.**. On dispose alors des méthodes *lock()* et *unlock()* pour prendre ou lâcher le verrou.

En CSharp : Une classe de synchronisation *Mutex* permet la même chose. Les méthodes *WaitOne()* et *Close* assure la prise du verrou et son relâchement (cf. code 6 p.31).

3.2 Sémaphore

Un sémaphore est une variable protégée (ou un type de données abstrait). C'est la méthode la plus couramment utilisée pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente. Le sémaphore utilise trois méthodes pour fonctionner :

- Init() : Initialise le sémaphore
- P() : Permet d'attendre qu'une ressource soit disponible
- V() : Permet de rendre la ressource disponible à ceux qui l'attendent

Un sémaphore est en réalité un mutex mais qui intègre un compteur. Un exemple qui montre bien leur utilisation est le "dîner des philosophes".

En C avec les threads POSIX : Il existe une librairie pour utiliser des sémaphores avec les threads POSIX et dont les fonctions sont définies dans le fichier *semaphore.h*. On crée alors un objet *sem_t* et on dispose des fonctions *sem_init()* pour initialiser le sémaphore, de *sem_wait()* pour le P() et de *sem_post()* pour le V().

En Java : Il faut utiliser la classe *Semaphore* présente dans *java.util.concurrent.**. Les méthodes P() et V() sont représentées respectivement par les méthodes *acquire()* et *release()*.

3.3 Section critique

Il arrive parfois qu'au sein d'un programme multi-thread, il soit nécessaire qu'un seul thread exécute à la fois une portion de code, afin de ne pas perturber l'utilisation d'une variable partagée. Par exemple, si au sein d'une tâche, on a une instruction qui met à jour la valeur d'une variable partagée, suivie d'une instruction utilisant cette variable, il faut être sûr que la valeur de la variable soit bien celle que l'on a initialisée avant de l'utiliser, et qu'elle n'as pas été mise à jour entre temps par un autre processus.

Il est possible de gérer les sections critiques en utilisant des mutexs sur les variables sensibles. Cependant certains langages ont des mécanismes plus simples pour leur mise en œuvre.

En Java : une section critique peut être définie en utilisant le bloc d'instructions *synchronized(...){section critique}*, ou en plaçant la primitive *synchronized* dans l'entête d'une méthode.

En C avec OpenMP : il faut utiliser la clause *#pragma omp critical {section critique}*, et inclure le code de la section critique au sein du bloc.

3.4 Modèle producteur-consommateur

Ce modèle consiste à ce qu'un consommateur soit notifié [8] à chaque mise à jour d'une ressource par un des producteurs. Ainsi, le consommateur verrouille la ressource partagée et se met en attente, puis lorsqu'il est réveillée, il consomme la ressource partagée et effectue ses traitements avant de se remettre en attente.

8. Prévenu

Algorithm 5 Algorithme du producteur

```
1: Créer un document D
2: Verrouiller F
3: Ajouter D à la fin de la file F
4: Envoyer un signal par F au processus consommateur
5: Déverrouiller F
```

Algorithm 6 Algorithme du consommateur

```
1: Verrouiller F
2: while Il y a des consommateurs do
3:     Attendre signal de F
4:     for all Elément E de F do
5:         Consommer E
6:         Effacer E de la file F
7:     end for
8: end while
9: Déverrouiller F
```

En Java : Ce système peut être facilement implémenté en utilisant le bloc d'instructions *synchronized(ressource){...}*, qui permet de verrouiller une ressource. On peut alors utiliser les méthodes *wait* et *notify* sur la ressource pour attendre et envoyer un message.

3.5 Autres bibliothèques de synchronisation

Les outils de synchronisation de base comme les mutex ou les sémaphores ne suffisent plus à créer des programmes ou mettre en œuvre des moyens de synchronisation très poussés sans rentrer dans les "usines à gaz" assez rapidement. De plus, il y a toujours un risque de créer des dead-lock [9]. Les dead-lock arrivent lorsque qu'au même instant deux fonctions rentrent en concurrence des mêmes mutex. Restant ainsi bloquée indéfiniment.

3.5.1 Bibliothèque Java

Il existe un package nommé «java.util.concurrent» qui permet l'accès atomique à une même ressource, comme une liste chaînée par exemple.

Entre outre, avec la classe *LinkedBlockingQueue* elle prend en charge l'attente d'un élément dans la *queue* lorsque l'on appel la méthode *take*. Lorsque l'on dépose un élément dans la *queue* avec *offer(Object)* la méthode *take* est débloquée. Rien que cette classe permet de faciliter la programmation d'une *queue* de type producteurs-consommateurs.

3.5.2 Bibliothèque C Sharp

Les exemples sont disponibles dans l'ANNEXE A p.30.

9. Dead-lock : processus qui ne fait plus rien, bien souvent bloqué par un autre processus

Monitor : En C# il existe la bibliothèque *System.Threading*. Celle-ci permet de créer des sémaphores à partir de l'objet *Monitor*. Pour se synchroniser il faut prendre le verrou avec *"Monitor.Enter(this)"* et *"Monitor.Exit(this)"* pour le relâcher(cf. code 3 p.30). Il est aussi possible de définir si le verrou est déjà récupéré via la méthode *"Monitor.TryEnter(this)"*. Cette fonction retourne faux lorsque le verrou est récupéré par quelqu'un d'autre. Il est possible d'utiliser la même fonction avec un paramètre supplémentaire de timeout. Ainsi, le verrou peut tenter d'être pris pendant un certain temps. Il s'agit d'une fonction bien utile lorsque l'on veut éviter certains dead-lock (cf. code 4 p.30).

ReaderWriterLock : La classe `ReaderWriterLock` permet de réaliser une synchronisation en lecture ou en écriture. Il est possible de lire de façon concurrente par rapport à d'autres lectures. Par contre, il n'est pas possible de lire ni écrire pendant une synchronisation en écriture.

La méthode est la suivante. On crée une classe *«ReaderWriterLock»*. Lorsque l'on souhaite accéder à une variable (ex : une liste) on appel la méthode *AcquireReaderLock* de cette classe pour prendre le verrou en lecture. Si on souhaite écrire, on utilisera *AcquireWriterLock*. Après la lecture ou l'écriture, l'appel de *ReleaseReaderLock* rend le verrou.

Cette classe est très utile lorsque l'on souhaite réaliser un programme avec plusieurs lecteurs et peux d'écrivains.

4 Bilan

4.1 Technique

Ce projet nous a permit de découvrir et d'approfondir nos connaissances de la programmation multitâches et programmation concurrente.

Les différents tests de performances effectués vont nous permettre à l'avenir d'argumenter nos futurs choix de conception et choix du langage de programmation à utiliser.

De même, nous avons pu remarquer l'importance des JVM utilisées et des paramètres de compilation à utiliser pour avoir un pourcentage d'exploitation de la machine maximum. On peut très bien avoir une machine très puissance, fonctionnant moins vite qu'une machine standard à cause de mauvais paramètres de compilation.

Nous avons aussi constaté qu'en utilisant des bibliothèques du type *concurrent* et *JSR166*, il était possible d'améliorer considérablement un programme pouvant être parallélisé. Celui-ci devenant meilleur que le langage C.

4.2 Humain

Ce projet nous a apporté de la clarté dans utilisation des outils de synchronisation de différents langages. Ce rapport est une véritable documentation réutilisable dans votre future expérience professionnelle.

Ce projet nous a fait toucher du doigt un certain nombre de choses que nous n'avions pas conscience lors de nos études. La synchronisation reste encore très difficile à manipuler, les bibliothèques de synchronisation sont là pour nous aider à programmer de façon concurrente plus aisément et à faire de nos futurs programmes parallèles des programme efficaces. Mais ce n'est pas encore assez pour résoudre de simples problèmes d'ordonnancement.

La programmation concurrente et l'utilisation de programmes multi-threadés devient de plus en plus exploitées par les processeurs actuels, gagent de performances lorsque ceux-ci sont bien exploités. Nous pensons donc être en mesure dans notre prochaine expérience professionnelle de fournir le maximum de ce que nous connaissons dans le domaine du multitâche pour enrichir les compétences de la société qui nous emploiera.

Conclusion

En conclusion, nous pouvons dire que ce projet nous a permis d'étoffer nos connaissances dans les mécanismes de programmation multi-thread, à travers les tests de multiples bibliothèques dans divers langages de programmation sans oublier les différents mécanismes de synchronisation existant. Grâce à cela, nous avons pu nous forger une expérience qui ne manquera pas de nous être utile plus tard.

Cependant, ce projet ne fut pas des plus aisé pour nous, car il était impossible pour nous d'obtenir une liste exhaustive des tests à réaliser compte tenu de la diversité des plates-formes existantes et des outils de programmation multi-thread, ainsi que de nos moyens techniques disponibles.

Toutefois, nous avons pu établir un bon récapitulatif des performances que peuvent fournir les outils que nous avons testés.

Bibliographie

Liens sur OpenMP :
http://openmp.org/wp/
http://www2.lifl.fr/west/courses/cshp/openmp.4.ps.gz
htpp://www.idris.fr/data/cours/parallel/openmp/OpenMP_cours.pdf
Liens sur TBB :
http://www.threadingbuildingblocks.org/
Liens sur Threads Posix :
http://www.dil.univ-mrs.fr/~massat/ens/docs/thread-sem.html
Liens sur Java :
http://www.cui.unige.ch/java/JAVAF/signaux.html
http://rom.developpez.com/java-synchronisation/
http://www.javafr.com/tutoriaux/JAVA-SYNCHRONISATION_540.aspx
http://www.cui.unige.ch/java/JAVAF/signaux.html
http://www.cui.unige.ch/java/JAVAF/signaux.html

Liens sur CSharp :
http://fr.wikibooks.org/wiki/Programmation_C_sharp/Threads_et_synchronisation#
Autres_outils_de_synchronisation
http://en.csharp-online.net/Building_Multithreaded_Applications%E2%80%94The_
Issue_of_Concurrency
Autres :
http://www.robot.jussieu.fr/webadmin/UserFiles/File/clady_homepage/M2/Initiation_
PROCESS.pdf
info.usherbrooke.ca/GabrielGirard/cours/archives/hiver-2008/ift630/projets-hiver-20
documents/projet630.doc

A La synchronisation en CSharp

Exemple : Dans l'exemple ci-dessous on peut voir deux threads en concurrence qui lance la même fonction `longueTache` sur l'objet prog. Dans les paragraphes suivants, des exemples de fonction synchronisée.

Listing 1 – *Exemple de synchronisation en CSharp*

```
using System;
using System.Threading;
class Program{
  public static void Main(string[] args){
    Program prog = new Program();
    Console.WriteLine("Hello World!");
    Thread thread = new Thread(prog.longueTache);
    thread.Start();

    Thread thread2 = new Thread(prog.longueTache);
    thread2.Start();

    thread.Join();
    thread2.Join();

    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
  }
}
```

Synchronisation de base : Dans cet exemple on utilise la primitive de base `lock`.

Listing 2 – *Utilisation de la primitive lock*

```
class Program{
  public void longueTache(){
    int a=100;
    int b=10;
    lock(this){
      //say we do something here.
      Thread.Sleep(1000); // attend 1 seconde
      int c=a/b;
    }
  }
}
```

Synchronisation avec `Monitor` : Dans cet exemple on utilise la classe `Monitor`.

Listing 3 – *Utilisation de la classe Monitor*

```
class Program{
  public void longueTache(){
    int a=100;
    int b=10;
    Monitor.Enter(this);
    //say we do something here.
    Thread.Sleep(1000); // attend 1 seconde
    int c=a/b;
    Monitor.Exit(this);
  }
}
```

Synchronisation avec `Monitor` et tentative de prendre le verrou : Dans cet exemple au moment de prendre le verrou, on test si le verrou n'est pas en cours d'utilisation. S'il est en cours d'utilisation, on affiche `Echec` sinon `Msg`

Listing 4 – *Utilisation de la méthode TryEnter pour prendre le verrou*

```
class Program{
  public void longueTache(){
    for(int i = 0; i < 1000000; i++){
      int id = Thread.CurrentThread.ManagedThreadId;
      if (Monitor.TryEnter(this, 800)){
```

```
            Thread.Sleep(1000); // attend 1 seconde
            Console.Write(id + " : Msg "+i + "\n");
            Monitor.Exit(this);
          }else{
            Console.Write(id + " : Echec "+i + "\n");
          }
        }
      }
}
```

Synchronisation avec `Mutex` : Dans cet exemple on utilise la classe `Mutex`.

Listing 5 – *Utilisation de la classe Mutex*

```
class Program{
  public void longueTache(){
    int a=100;
    int b=20;
    Mutex firstMutex = new Mutex(false);
    firstMutex.WaitOne();
    //some kind of processing can be done here.
    Thread.Sleep(1000); // attend 1 seconde
    int x=a/b;
    firstMutex.Close();
  }
}
```

Synchronisation readWriteLock Dans cet exemple on utilise la classe `ReaderWriterLock`. Celle-ci permet de synchroniser en lecture ou en écriture une partie du code. Lors d'une lecture toutes les instructions accèdent à la section critique. Tant dis que l'écriture provoque la synchronisation de l'accès à la section critique.

Listing 6 – *Utilisation de la classe Mutex*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace ReaderWriterSample {
    class MyClass{
        private static ReaderWriterLock listlock =
            new ReaderWriterLock();
        private static List<string> mylist = new List<string>();


        public void DoSomeWriting() {
            listlock.AcquireWriterLock(-1);
            Console.WriteLine("Writing...");
            mylist.Add(DateTime.Now.Ticks.ToString());
            Console.WriteLine("Count is " + mylist.Count.ToString());
            listlock.ReleaseWriterLock();
        }

        public void DoSomeReading() {
            listlock.AcquireReaderLock(-1);
            Console.WriteLine("Reading...");
            Console.Write("List: (count " + mylist.Count.ToString()
                + ")" );
            foreach (string item in mylist) {
                Console.Write(item);
                Console.Write("-");
            }
            Console.WriteLine();
            listlock.ReleaseReaderLock();
        }
    }

    class Program{
        static void Main(string[] args) {

            MyClass inst = new MyClass();

            Random rnd = new Random();
            for (int i = 0; i < 20; i++ ) {
                Console.WriteLine(i);
                if (i % 4 == 0) {
                    Thread t = new Thread(inst.DoSomeWriting);
                    t.Start();
                }
```

```
                else {
48                  Thread t = new Thread(inst.DoSomeReading);
                    t.Start();
                }
            }
        }
53      }
    }
```