Placement des threads sur les cœurs: *Threads affinity*

Sid Touati
Professeur à l'université Côte d'Azur

Le but de ces exercices pratiques est de vous montrer comment placer les threads sur les divers cœurs, ainsi que leur impact sur les performances. Afin de pouvoir faire ce TP, vous devez installer les logiciels suivants sur votre machine personnelles :

1. Linux (64bits).
2. Compilateurs C : `gcc` et `gcc`.
3. Librairie `libgomp` (normalement c'est installé par défaut avec gcc, mais vérifiez).
4. Logiciel de traitement statistique R : `https://www.r-project.org`
5. `R-Studio` (version gratuite) : `https://rstudio.com/products/rstudio/download`

Prenons comme example un code de multiplications de matrices, connu comme étant un code massivement parallèle. Le code C est donné dans un fichier joint. Vous pouvez le modifier à votre guise, notamment pour augmenter la taille des matrices.

Le code fourni est une implémentation d'une version parallèle avec OpenMP, son exécution créera plusieurs *threads*. Le nombre de threads OpenMP durant l'exécution ne doit pas dépasser le nombre de cœurs logiques du processeur afin de ne pas créer des situations de concurrence inutile au sein du même cœur. Vérifiez le nombre de cœurs sur votre PC en examinant le fichier `/proc/cpuinfo`.

1 Mesures des performances

Le benchmark que vous devrez exécuter et mesurer les performances est le code appelé `matrixmatrixmultiply.c`.

1.1 Compiler le code C

Vous pouvez compiler le code C en utilisant les commandes suivantes :

```
gcc -O2  -fopenmp ./matrixmatrixmultiply.c -o ./matrix-gcc
icc O2 -openmp ./matrixmatrixmultiply.c o ./matrix-icc
```

Les deux commandes ci-dessus créent deux exécutables différents, générés par deux compilateurs différents. Ces deux fichiers exécutables utilisent des librairies OpenMP différentes. Vérifiez cela avec la commande `ldd`.

1.2 Exécuter les benchmarks en variant le nombre de threads

Le nombre de threads créés à l'exécution peut être précisé avant l'exécution d'une application OpenMP. Pour cela il faut utiliser des variables d'environnement comme ceci, où je prends comme exemple la création de 4 et 5 threads :

```
export OMP_NUM_THREADS=4
./matrix-gcc

export OMP_NUM_THREADS=5
./matrix-icc
```

Pour mesurer les temps d'exécution, vous pouvez utiliser la commande `time` comme suit :

```
export OMP_NUM_THREADS=4
time ./matrix-icc
```

La commande `time` permet de reporter plusieurs mesures, qui sont le temps réel d'exécution, le temps d'exécution en mode *user* ainsi que le temps d'exécution en mode système. Dans ce TP, nous nous intéressons au temps réel. La commande suivante permet de filtrer la sortie de time pour ne capturer que les données qui nous intéressent, redirigées vers un fichier texte appelé `toto.data` par exemple.

```
 export OMP_NUM_THREADS=4
/usr/bin/time -f "%e" -o toto.data  ./matrix-gcc
```

Vérifiez que le fichier `toto.data` contient bien les mesures du temps d'exécution.

Afin de répéter l'exécution d'un benchmark plusieurs fois (ici 50 fois) et sauvegarder les mesures dans un fichier `bench.data`, la commande suivante vous donne un exemple :

```
export OMP_NUM_THREADS=4
for i in ‘seq 1 50‘ ; do /usr/bin/time -f "%e" -o toto ./matrix-gcc ; cat toto >> bench.data ; done
```

Je vous propose de réaliser 50 mesures pour les deux exécutables (`matrix-gcc, matrix-icc`), en fixant comme nombre de threads le nombre de cœurs logiques de votre processeur. Sauvegardez les mesures dans deux fichiers séparés `bench1.txt` et `bench2.txt`, chacun contient 50 nombres. La section suivante explique comment analyser ces data et vérifier si les performances sont stables.

2 Stabilité des performances d'un programme

L'analyse des performances se fera avec le logiciel R, dont l'interface graphique est `R-Studio`. R peut être utilisé en ligne de commande aussi. Après avoir lancé l'interface graphique `R-Studio`, commencez par charger manuellement quelque librairies indispensables en tapant les commandes suivantes dans la console R (installez ces librairies dans R si nécessaire) :

```
library(stats)
library(graphics)
library(vioplot)
```

Maintenant, vous pouvez charger vos données stockées dans les deux fichiers `bench1.data` and `bench2.data` en tapant les commandes suivantes dans la console R :

```
X <- read.csv("bench1.data", header=F)$V1
Y <- read.csv("bench2.data", header=F)$V1
```

Maintenant, deux ensembles de données `X` et `Y` contiennent les données collectées (les temps d'exécution des deux benchmarks, répétés 50 fois chacun). Si vous souhaitez calculer les premières statistiques empiriques de vos données, utilisez les commandes suivantes dans la console R :

```
summary(X)
summary(Y)
```

Que déduisez vous ? Est ce qu'il y a une variation sensible des performances ? Une stabilité ?

2.1 Boxplots

Maintenant, visualisons les données en utilisant les boxplots comme ceci :

```
boxplot(X,Y,names=c("bench1","bench2"),col=c("blue","pink"),ylab="secondes", main="Temps d'exécution")
```

Les boxplots permettent to visualiser rapidement le maximum, minimum, quartiles et médianes. Quelles conclusions portez vous ? Concernant la stabilité des performances, sur la comparaison des performances, etc. Est ce que `bench1` est plus rapide que `bench2` ? est-ce l'inverse ?

3 Comment fixer le placement des threads sur les cœurs avec Linux

L'exécution des benchmarks dans l'exercice précédent crée plusieurs threads. Par contre, rien n'indique comment les threads sont placés sur les cœurs du processeur. Le système d'exploitation (OS) les ordonnance librement selon ses propres politiques de placement de threads. Dans cet exercice, nous allons apprendre comment imposer le placement des threads selon des stratégies proposées par OpenMP et non pas par l'OS. Le placement de threads est appelé *thread affinity* ou *thread binding* dans la littérature technique en anglais.

3.1 Placement des threads avec `gcc`

Concernant la librairie `libgomp` utilisée par `gcc`, la variable d'environnement `GOMP_CPU_AFFINITY` permet d'indiquer avec précision sur quel cœur **logique** doit s'exécuter chaque thread dans l'ordre de création. Par exemple, la commande suivante :

```
export OMP_NUM_THREADS=4
export GOMP_CPU_AFFINITY="0 1 2 3"
 ./matrix-gcc
```

placera les 4 threads sur les cœurs logique 0, 1, 2, 3 dans l'ordre. Cette variable d'environnement permet d'explorer plusieurs placements de threads possibles. Je vous propose d'expérimenter deux options :

1. *Option 1* : placer les threads consécutifs sur des cœurs consécutifs qui ont des caches partagés si possible.
2. *Option 2* : placer les threads consécutifs sur des cœurs physiques éloignés qui ne partagent pas de caches.

Pour connaître l'architecture précise de votre processeur, je vous invite à consulter le document *Comment analyser l'architecture matérielle d'un ordinateur sous Linux ?*

Faites des mesures de performances avec 50 répétitions, et comparez entre les différentes options vis à vis de celle de l'OS. Utilisez la visualisation graphique avec les boxplots pour faire vos comparaisons et prendre des conclusions.

3.2 Placement des threads avec `icc`

Le compilateur d'intel `icc` utilise une autre librairie OpenMP. Les options de placement de threads sont spécifiées via une autre variable d'environnement appelée `KMP_AFFINITY`. Cette variable d'environnement peut être utilisée en spécifiant le type de placement de thread souhaité, avec des noms comme `compact`, `scatter`, `none`. Le placement `none` donne la liberté à l'OS de placer les threads où il veut. Voici un exemple d'usage :

```
export OMP_NUM_THREADS=4
export KMP_AFFINITY=compact
 ./matrix-icc
```

Faites des mesures de performances avec 50 répétitions, et comparez entre les différentes options `KMP_AFFINITY=compact` et `KMP_AFFINITY=scatter` vis à vis de celle de l'OS (`KMP_AFFINITY=none`) et celles de `gcc`. Utilisez la visualisation graphique avec les boxplots pour faire vos comparaisons de performances et prendre des conclusions.

Explications sur le fonctionnement de `compact`, `scatter` : Voici comment se comportent les stratégies de placement de threads proposées par Intel :

1. `KMP_AFFINITY=compact` permet de placer le prochain thread créé sur le cœur libre le plus "proche" physiquement du dernier cœur alloué. [1] Cette stratégie a pour objectif favoriser le partage des caches entre les cœurs adjacents, afin que deux threads consécutifs dans la création qui accèderaient aux mêmes données puissent bénéficier potentiellement d'un cache partagé.
2. `KMP_AFFINITY=scatter` permet de placer le prochain thread créé sur le cœur libre le plus "éloigné" physiquement du dernier cœur alloué. C'est la stratégie inverse de `KMP_AFFINITY=compact`. Cette stratégie a pour objectif de ne pas favoriser le partage les caches entre les cœurs, afin que chaque thread bénéficie potentiellement d'un plus grand cache à lui tout seul. Cette stratégie est préconisée lorsque les threads ne partagent pas de données.

Autres possibilités pour `KMP_AFFINITY` : D'autres valeurs pour `KMP_AFFINITY` sont possibles, comme :

1. `KMP_AFFINITY=explicit` permet de spécifier manuellement le placement des threads en utilisant la variable `GOMP_CPU_AFFINITY`) qui liste l'ordre des cœurs logiques.
2. `KMP_AFFINITY=disabled` permet de désactiver le placement explicit des threads sur les cœurs, seul l'OS peut placer les threads selon ses propres méthodes d'ordonnancement sur CPU.
3. `KMP_AFFINITY=verbose` imprime des informations détaillées sur l'affinité pendant l'exécution.

Pour aller plus loin : La librairie OpenMP d'Intel offre des fonctions pour fixer le placement des threads sur les cœurs par programme durant l'exécution, ou récupérer le numéro de cœur sur lequel s'exécute un thread. Les fonctions en question sont `kmp_set_affinity` et `kmp_get_affinity`. Je ne connais pas un équivalent avec `libgomp`.

1. En fait, l'architecture du processeur modélise l'adjacence entre les cœurs avec un arbre, permettant de détecter l'adjacence entre les cœurs. Cet arbre peut être retrouvé en analysant la structure hiérarchique des répertoires dans le système de fichiers Linux `/sys/devices/system/cpu/cpu?/topology`.