TP de Systèmes d'exploitation
François Cayre / francois.cayre@grenoble-inp.fr

Multiplication parallèle de matrices

Notions abordées : *threads*, synchronisation, granularité, localité mémoire.

Fonctions utilisées : `pthread_create(3)`, `pthread_join(3)`, `pthread_mutex_lock(3)`, `pthread_mutex_unlock(3)`.

1 Introduction de la notion de granularité

Le terme courant de la multiplication de deux matrices A et B de dimensions respectives (L,N) et (N,C) est défini par :

$$P[l][c]=\sum_{k=1}^{N} A[l][k]\times B[k][c] \quad .$$

Du point de vue algorithmique, cette expression possède une complexité temporelle en O(CLN). Strassen [1] a montré comment multiplier deux matrices avec une complexité un peu inférieure, un résultat amélioré par Coppersmith et Winograd [2], mais nous nous contenterons de l'expression ci-dessus pour ce TP.

Cette expression ne dit rien de la manière dont il conviendrait de l'implanter sur une machine possédant plusieurs cœurs physiques d'exécution (CPUs). Lors du TP précédent, vous avez implanté la version séquentielle de ce calcul (réalisée en n'utilisant qu'un seul CPU). Cette version séquentielle est donc celle utilisée dans la session suivante, tirée du TP précédent :

```
var N = 5000, A = rand(20,N,"int10"), B = rand(N,20,"int10") ;
var P = A*B ;
```

Dans ce TP, nous nous proposons d'implanter une multiplication parallèle de deux matrices, qui sera à coder dans le greffon `op_mmult_par.c` du répertoire `plugins` du TP précédent, et dont nous vous fournissons une amorce pour démarrer sereinement. On pourra donc comparer la version séquentielle (réputée fournir un résultat juste) avec la version parallèle à l'aide de :

```
var Ppar = mmult_par(A,B) ;
P==Ppar ;
```

On désigne par le terme de *granularité* la quantité de travail confiée à un *thread*. Si le *thread* a relativement peu de choses à faire, on dit que la granularité est *fine*. La granularité d'une implantation parallèle est la conséquence directe de la conception de la parallélisation. Dans ce TP, nous discutons la notion de granularité en comparant différentes conceptions pour la parallélisation.

2 Travail à réaliser

2.1 Multiplication parallèle sans synchronisation

Vous implanterez la multiplication parallèle de deux matrices avec la granularité la plus fine possible : pour chaque terme courant de la matrice résultat, vous lancerez N *threads*, chacun responsable de l'incrémentation de P[l][c] avec le produit A[l][k]xB[k][c]. Cette première étape est à réaliser exclusivement dans la fonction `compute_coeff`.

Expliquez les résultats obtenus sans synchronisation.

2.2 Ajout d'une synchronisation

Nous vous proposons de comparer deux stratégies de synchronisation. Nous vous fournissons une première stratégie fondée sur le *Compare-and-Swap*, qui utilise l'instruction assembleur `cmpxchg` : `SYNC_COMPARE_AND_SWAP`. Comparez les résultats obtenus avec et sans synchronisation.

Dans une second étape, vous implanterez la stratégie de synchronisation fondée sur les mutex POSIX, appelée `SYNC_MUTEX` dans le code fourni. Comparez les résultats obtenus.

Comparez la facilité de programmation des deux stratégies de synchronisation.

2.3 Deux autres conceptions de la parallélisation

Nous vous fournissons deux greffons supplémentaires : `op_mmult_par2.so` et `op_mmult_par3.so`.

Le premier distribue la charge de travail en confiant le calcul de L/CPUs lignes de P à chaque *thread*. Comparez les temps de calcul de `op_mmult_par.so` et `op_mmult_par2.so`. Concluez sur (1) la notion de granularité, et (2) la conception globale de `op_mmult_par.so`. Pouvait-on rendre `op_mmult_par.so` plus rapide en conservant la même granularité ? Si oui, comment ? Et y avait-on alors malgré tout intérêt ? Si non, pourquoi ?

Le second greffon supplémentaire, `op_mmult_par3.so`, procède exactement de la même manière que le premier, à la différence près que la multiplication est adaptée pour être effectuée sur la *transposée* de B. On rappelle que dans notre application, les coefficients des matrices sont stockés contigûment une colonne après l'autre. Comparez les temps de calcul et concluez sur la localité des accès à la mémoire.

Concluez cette partie en donnant les critères d'une parallélisation réussie.

2.4 Application au calcul scientifique

On nous dit dans l'oreillette que ceux d'entre vous ayant suivi le cours de *Calcul scientifique* de M. Rivet seront en mesure d'apprécier ceci à sa juste saveur…

Jusqu'ici, nous avons testé nos trois parallélisations en utilisant des entiers entre 0 et 9 (la fonction `rand` était appelée avec `"int10"`). Sélectionnez maintenant l'une des trois versions parallèles de la multiplication, et expliquez ce que vous observez en travaillant à présent sur des flottants entre 0 et 1 :

```
var N = 5000, A = rand(20,N,"uniform"), B = rand(N,20,"uniform") ;
var P = A*B ;
var Ppar = mmult_par(A,B) ;
P==Ppar ;
```

Concluez sur l'égalité numérique de deux matrices.

Serait-il possible de mettre en place une stratégie globale de parallélisation des opérations d'algèbre linéaire sur une machine disposant de plusieurs CPUs et se partageant la mémoire centrale ? Si oui, esquissez-en les grandes lignes. Si non, expliquez pourquoi.

3 Compléments

3.1 Barrières

Vous remarquerez que notre conception de la parallélisation du calcul, indépendamment de la granularité, implante quelque-chose ressemblant à une *barrière*. Une barrière consiste à lancer des *threads* pour exécuter une partie parallèle du code, et attendre la terminaison de ce calcul parallèle pour revenir à une partie non parallélisable (séquentielle) du code. Et on enchaîne ainsi à la suite les parties parallélisables et séquentielles du code. C'est une conception très simple de la parallélisation (de bien plus compliquées existent), qui trouve de surcroît de très nombreuses applications en calcul scientifique. L'implantation des *threads* POSIX sous Linux propose le type `pthread_barrier_t` pour faciliter la vie du programmeur (on attend encore pour MacOS). Cette distinction entre parties parallélisables ou non du code culmine dans la (très théorique…) Loi d'Amdahl.

3.2 Loi d'Amdahl

En ignorant absolument, entre autres, toutes les caractéristiques physiques de la machine et en ne considérant que le code à exécuter, G. Amdahl propose de calculer le gain théorique maximal atteignable en parallélisant du code [3].

Soit P la durée (en proportion) d'exécution de la partie parallélisable du code, et (1-P) la durée (en proportion) d'exécution de sa partie séquentielle. En fonction du nombre de CPUs, le gain de temps maximal théorique est d'un facteur G(P,CPUs) :

$$G(P,CPUs)=\frac{1}{1-P+\frac{P}{CPUs}}.$$

En pratique, les caractéristiques physiques de la machine et la conception du système d'exploitation ont un impact non négligeable (vitesse de changement de contexte, performance de l'ordonnanceur, débit du bus mémoire, motif d'accès à la mémoire, taille des caches, *etc.*) Ainsi, du code parallélisable à P=50 % peut-il espérer, avec un grand nombre de CPUS et au mieux, diviser son temps de calcul par (seulement) deux (au maximum).

4 Références

[1] V. Strassen, « *Gaussian Elimination is not Optimal* », Numer. Math., 13, pp. 354-356, 1969.

[2] D. Coppersmith et S. Winograd, « *Matrix Multiplication via Arithmetic Progression* », Journal of Symbolic Computation, 9(3), pp. 251-280, 1990.

[3] G. M. Amdahl, « *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities* », Proc. 30th AFIPS Conference, pp. 483-485, 1967.

Bon travail !