julien.braine@ens-lyon.fr,mateusz.skomra@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

ASR1, TP10 : La hiérarchie mémoire

Le but de ce TP est de mesurer expérimentalement la performance de la hiérarchie de caches de la machine. Pour cela, on va écrire intentionnellement un programme dont le comportement mémoire aura une mauvaise localité, de façon à provoquer un maximum de défauts de cache.

En chronométrant l'exécution du programme, et en divisant la durée obtenue par le nombre d'accès, on obtiendra une mesure approximative de la latence mémoire.

1 Préliminaires : un programme avec une mauvaise localité spatiale

On va faire exprès de parcourir une structure de données en s'assurant que deux accès successifs ne sont jamais à des adresses voisines. L'idée est de forcer une mauvaise localité spatiale, de façon à ce que chaque accès mémoire ait une latence maximale.

Ce principe est illustré par le schéma ci-dessous. Un parcours séquentiel accède à la mémoire avec une bonne localité spatiale, donc la mise en cache «fausse» les temps d'accès. À l'inverse, un parcours «dans le désordre» a de meilleures chances de provoquer un «vrai» accès à chaque donnée, donnant ainsi une meilleure idée de la latence mémoire.

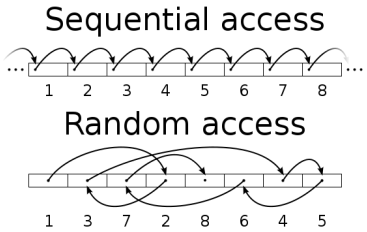

FIGURE 1 – Accès séquentiel vs Accès arbitraire (*Random Access*). Dessin tiré de wikipedia : `https://en.wikipedia.org/wiki/Random_access` Allez lire cette page si la distinction entre les deux modalités d'accès ne vous apparaît pas clairement.

Comme illustré par le schéma, une façon simple pour faire des accès arbitraires est de parcourir une liste chaînée dont on a mélangé l'ordre des éléments. C'est ce que vous allez faire dans les exercices suivants. Pour vous simplifier la vie, vous allez utiliser des indices de tableau plutôt que des pointeurs, mais l'idée est la même : toute la liste chaînée va être implémentée par un grand tableau d'entiers, et la valeur de chaque case servira de pointeur de chaînage.

Exercice 1: Téléchargez et ouvrez l'archive correspondant à ce TP. Constatez que la fonction `benchmark` de `listwalk.c` alloue dynamiquement un tableau de N éléments, initialise chaque case tab[i] à la valeur i, puis parcourt `R` fois cette liste chaînée.

Ce programme est aussi emballé dans ce qu'il fait pour le chronométrer au moyen de la fonction `clock()`.

Compilez puis lancez ce programme. Il s'agit pour le moment d'un accès séquentiel.

Exercice 2: Modifiez le programme donné pour qu'il mélange aléatoirement l'ordre des éléments du tableau.

Remarques
- Mathématiquement parlant, il s'agit de faire une permutation aléatoire.
- Pour faire des tirages aléatoires utilisez la fonction random() ou la fonction rand().
- Pour obtenir un comportement différent à chaque exécution, initialisez votre générateur aléatoire (à l'aide des fonction srandom() ou srand()) avec comme paramètre l'heure courante, par exemple : srandom(clock())

Exercice 3: Fixez la valeur de N à 10 éléments, la valeur de R à 1. Modifiez le parcours pour qu'il ajoute l'affichage de la case visitée. Faites plusieurs exécutions.

Sauf coup de chance, certaines exécutions ne parcourent qu'une sous-partie des indices possibles, à cause du mélange aléatoire. Modifiez votre algorithme de mélange pour garantir que le parcours de la liste visitera toujours l'ensemble des indices avant de tomber sur un zéro.

Remarques
— Mathématiquement parlant, il s'agit de faire une *permutation circulaire de longueur N*.
— En récompense pour les gens qui lisent les consignes jusqu'au bout, la solution est sur wikipedia : `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#Sattolo.27s_algorithm`

Faites valider votre travail par un enseignant (mais attaquez la suite sans attendre).

2 Mesure du temps d'accès mémoire

Dans cette deuxième partie, vous allez faire varier le paramètre N, et chronométrer le temps mis par le programme pour parcourir la liste en entier. L'idée est la suivante : si la liste est assez petite pour tenir dans le cache L1 alors chaque accès sera un *cache hit* dans le L1, donc la durée du parcours sera environ $N \times$Latence(L1). Si la liste déborde dans le L2, on aura alors un *miss rate* non nul, et le temps moyen observé sera situé entre la latence du L1 et la latence du L2. Et ainsi suite.

Exercice 4: Supprimez l'affichage des indices pour ne garder dans votre boucle finale que le parcours de la liste proprement dit. Fixez N à 1000 et R à 10000, et chronométrez l'exécution de la boucle (avec clock(), voir remarques ci-dessous). Le programme, si vous n'avez pas tout cassé, affiche la taille de la structure (en octets) ainsi que le temps d'accès moyen (en nanosecondes).

Remarques
— La fonction clock() vous renvoie une valeur dont l'unité est arbitraire. Pour convertir cette valeur en secondes, il faut la diviser par une constante dont le nom est CLOCKS_PER_SEC. Pour obtenir des nanosecondes, il faut ensuite multiplier par 10^9.
— Vous aurez remarqué que le Makefile qui vous est fourni invoque GCC avec l'option −O3, c'est à dire le niveau d'optimisation maximum. L'idée est d'obtenir une boucle de parcours la plus efficace possible, pour que nos mesures de temps ne soient pas faussées par des instructions « parasites » mais reflètent fidèlement la performance mémoire. Plus précisément, on va faire l'hypothèse que la seule opération coûteuse dans la boucle est l'accès au tableau, et que les autres instructions sont «gratuites».

Exercice 5: Répétez l'expérience pour plusieurs valeurs de N et constatez que votre temps d'accès moyen varie considérablement. Augmentez N jusqu'à stabiliser la valeur observée : vous êtes en train de mesurer la latence de la mémoire principale !

Remarques
— Si vos temps d'exécution deviennent trop importants, par exemple au-delà de 30s, réduisez la valeur de R pour compenser.

Faites ensuite une sauvegarde de votre programme, par exemple dans un fichier listwalk−2.c

3 Exploration de la hiérarchie des caches CPU

Dans cette partie, vous allez caractériser plus solidement la taille et le temps d'accès des différents niveaux de cache de la machine. Pour cela, vous allez simplement automatiser la variation des paramètres N et R.

Exercice 6: Depuis votre fonction main(), vous pouvez maintenant invoquer benchmark() dans une boucle en augmentant la valeur de N à chaque itération.

Remarques
— Ne faites pas varier N arithmétiquement mais géométriquement, par exemple en augmentant sa valeur de 10%, ou de 25%, à chaque itération. C'est le seul moyen d'atteindre les grandes tailles en un temps raisonnable.
— À propos de temps d'exécution : pour les petites tailles on peut répéter R =100000 fois le parcours, mais pour les grandes tailles ça deviendrait interminable, donc il vous faudra réduire la valeur de R au fur et à mesure. Une heuristique possible pour cela est de détecter les cas où benchmark() a pris trop de temps à s'exécuter (par exemple, au-delà de 1s) et de réduire alors R de moitié pour la prochaine itération (à condition que R⩾2).

Exercice 7: Les colonnes de chiffres obtenues à l'exercice précédent sont pleines d'information, mais pas évidentes à interpréter. Pour mieux appréhender vos résultats, faites-en une visualisation graphique, par exemple à l'aide d'un tableur comme LibreOffice, ou d'un autre outil de votre choix (gnuplot, numpy,...). Configurez votre graphique pour utiliser une échelle logarithmique sur les deux axes, afin d'accentuer les variations.

— Quand vous copiez-collez vos données dans le tableur, vérifiez bien que celui-ci interprète correctement vos nombres à virgule.

Exercice 8: En interprétant le graphique, répondez aux questions suivantes :

— combien de niveaux de cache sont présents sur la machine ?
— pour chaque niveau, quelle est sa taille ? son temps d'accès ?
— quelles est le temps d'accès de la mémoire principale (DRAM) ?

Comparez les tailles de caches que vous avez estimées avec celles indiquées par la commande `lscpu`.

4 Bonus : Multiplication de matrices

On s'intéresse maintenant à la multiplication de matrices, opération extrêmement fréquente en traitement d'image ou en calcul scientifique, et pour laquelle on doit être particulièrement attentif en terme de performance.

Rappel, la multiplication de deux matrices *carrées* est définie comme suit :

$$AB_{ij} = \sum_{k=1}^{N} A_{ik}.B_{kj}$$

En C, les tableaux à deux dimensions sont rangés en mémoire de façon à ce que toutes les cases d'une même ligne se suivent. La syntaxe `T[i][j]` désigne le j-ème élément de la i-ème ligne de T.

On vous donne ci-dessous les 6 versions possibles de ce programme, obtenues par permutations des boucles. Chaque version est identifiée par l'ordre d'imbrication des boucles. Toutes ces versions sont fonctionnellement équivalentes. On va voir qu'en terme de performance, il n'en est rien.

Exercice 9: Copiez-collez les 6 versions de ce programme (emballées du nécessaire (`main`, des déclarations de `A`, `B` et `C`, comme précédemment) et mesurez leur temps d'exécution sur des matrices 1000x1000.

Vous devriez pouvoir classifier ces 6 versions en trois paires de programmes aux performances similaires. Expliquez.

Exercice 10: Recommencez pour les valeurs de N variant de 100 en 100. Déduisez-en la technique de multiplication de matrices par blocs.

```
//version 1 (ijk)
for(i=0; i<N; i++)
    for(j=0; j<N; j++){
        r=0;
        for(k=0; k<N; k++)
            r += A[i][k]*B[k][j];
        C[i][j] += r;
    }
```

```
// version 2 (jik)
for(j=0; j<N; j++)
    for(i=0; i<N; i++){
        r=0;
        for(k=0; k<N; k++)
            r += A[i][k]*B[k][j];
        C[i][j] += r;
    }
```

```
// version 3 (jki)
for(j=0; j<N; j++)
    for(k=0; k<N; k++){
        r = B[k][j];
        for(i=0; i<N; i++)
            C[i][j] += A[i][k]*r;
    }
```

```
// version 4 (kji)
for(k=0; k<N; k++)
    for(j=0; j<N; j++){
        r = B[k][j];
        for(i=0; i<N; i++)
            C[i][j] += A[i][k]*r;
    }
```

```
// version 5 (kij)
for(k=0; k<N; k++)
    for(i=0; i<N; i++){
        r = A[i][k];
        for(j=0; j<N; j++)
            C[i][j] += r*B[k][j];
    }
```

```
// version 6 (ikj)
for(i=0; i<N; i++)
    for(k=0; k<N; k++){
        r = A[i][k];
        for(j=0; j<N; j++)
            C[i][j] += r*B[k][j];
    }
```