TP3 - Hiérarchie mémoire

Ce TP est partiellement inspiré du livre Dive into Systems (section 2.5 et chapitre 11).

Dans la pile ou dans le tas?

Créez un fichier `memoire.c` et recopiez le code suivant:

```
int n = 2, m=2;
int tab[n][m];

printf("# tab V1\n\n");
for(int i=0;i<n;i++){
    for(int j=0;j<m;j++)
        printf("%p\n",&tab[i][j]);
}

int **tab2 = malloc(sizeof(int *)*n);
for(int i=0;i<n;i++){
    tab2[i] = malloc(sizeof(int)*m);
}

printf("\n# tab V2\n\n");
for(int i=0;i<n;i++){
    for(int j=0;j<m;j++)
        printf("%p\n",&tab2[i][j]);
}
```

1. A votre avis, qu'affiche ce programme?
2. Quelles différences constatez-vous entre les deux tableaux `tab` et `tab2`?
3. Faites un dessin représentant la manière dont sont stockés les tableaux en mémoire

Attention au cache

Toujours dans votre fichier `memoire.c`, ajoutez les deux fonctions suivantes:

```
float moyenne_v1(int **mat, int n) {
    int i, j, total = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            total += mat[i][j];
        }
    }
    return (float) total / (n * n);
}

float moyenne_v2(int **mat, int n) {
    int i, j, total = 0;
    for (i = 0; i < n; i++) {
```

```
        for (j = 0; j < n; j++) {
            total += mat[j][i];
        }
    }
    return (float) total / (n * n);
}
```

Ces deux fonctions permettent de calculer la moyenne des valeurs d'une matrice (tableau à deux dimensions) d'entiers.

1. Quelle différence majeure constatez-vous entre ces deux algorithmes?
2. Si nous devions analyser le temps d'exécution "à la main", pensez-vous qu'un des deux algorithmes se démarqueraient?

Nous allons tenter de vérifier expérimentalement nos hypothèses. Nous allons comparer les performances en terme de temps d'exécution en utilisant les deux méthodes d'initialisation d'un tableau à deux dimensions vu à l'exercice précédant.

Avant de passer à la suite: Vous devez rédiger un compte-rendu expérimental des différentes mesures de performance à venir que vous exporterez au format pdf. Pour que l'expérience aie du sens, **on choisira une taille de tableau significative**. Par exemple, on choisira $n \geq 100$. Pour le remplissage de la matrice, des valeurs aléatoires ou arbitraires conviennent.

Dans un premier temps vous mesurerez le temps CPU utilisé à l'aide de la bibliothèque `time.h` et de la méthode `clock_gettime()` (voir tp2).

Nous allons ensuite mesurer le nombre d'instructions processeur utilisées par chaque algorithme. Pour cela, nous utiliserons l'outil `callgrind` de valgrind, vu au tp précédent. Pour rappel, voici les commandes à utiliser:

```
# Compiler le programme avec l'option g (debug)
gcc -g memoire.c -o memoire
# lancer valgrind avec notamment les options dump-line et simulate-cache à true
valgrind --tool=callgrind --dump-line=yes --simulate-cache=yes --collect-jumps=yes ./memoire
# annoter le fichier généré
callgrind_annotate --inclusive=yes --auto=yes callgrind.out.{pid} > rapport.txt
```

Tout d'abord, on relèvera le nombre d'instructions pour chaque fonction.

2. Dans quel cas constatez-vous une réelle différence de temps d'exécution?

Étudions maintenant l'impact du cache sur les performances. Sans rentrer dans les détails, le cache est une mémoire proche du CPU qui offre un temps d'accès beaucoup plus rapide que la RAM. L'idée est de stocker en cache certains blocs de données contiguës (par exemple ceux utilisés souvent) afin de pouvoir y accéder rapidement et limiter la communication avec la RAM. Lors d'un accès aux données, on regarde prioritairement dans le cache. Si la donnée s'y trouve (*cache hit*), elle est directement transmise au CPU. Sinon (*cache miss*), nous n'avons pas d'autre choix que d'aller la chercher en RAM.

Voici un extrait de la documentation de valgrind permettant d'analyser le rapport concernant l'utilisation du cache:

```
D cache reads (Dr, which equals the number of memory reads)
D1 cache read misses (D1mr),
LL cache data read misses (DLmr).
```

Déterminez les pourcentages de **cache misses** pour chacun des deux algorithmes et en fonction de la manière dont est défini le tableau (statique ou dynamique).

Parvenez-vous à mieux expliquer les différences de performance?

Remarque: En pratique, il y a plusieurs niveaux de cache. Cependant, c'est le cache de premier niveau (**L1**) qui nous intéressera le plus.

En conclusion

Prendre en compte les mécanismes de cache peut se révéler crucial pour optimiser les performances de vos algorithmes en pratique. Il est par exemple conseillé de faire attention à la manière dont sont définies vos structures de données ou encore de tenter d'accéder prioritairement à des blocs de données contiguës (accès par ligne et non pas par colonne de votre matrice). Pour aller plus loin, vous pouvez par exemple lire cet échange intéressant sur stackoverflow: https://stackoverflow.com/questions/763262/how-does-one-write-code-that-best-utilizes-the-cpu-cache-to-improve-performance.