TP 4 Parallélisme et instruction processeur

Exercice 1 (Mots binaires en parallèles)**.**

(1) Tout d'abord, assurez vous que vous avez bien récupéré le fichier mots_binaires.cpp ainsi que le fichier Makefile dans un même dossier. Ouvrez un terminal, rendez-vous dans le bon répertoire, et vérifiez que le fichier compile avec la commande make mots_binaires.

(2) Lancez l'exécution suivante ./mots_binaires 5, une liste de nombre apparaît :

Result : 1 5 10 10 5 1

Ce sont les nombres de mots binaires de taille 5 regroupés par leur nombre de bits à 1. On lit la suite de cette façon : il existe
— 1 mot binaire de taille 5 avec 0 bit à 1
— 5 mots binaires de taille 5 avec 1 bit à 1
— 10 mots binaires de taille 5 avec 2 bits à 1
— 10 mots binaires de taille 5 avec 3 bits à 1
— 5 mots binaires de taille 5 avec 4 bits à 1
— 1 mot binaire de taille 5 avec 0 bit à 1

(3) Ouvrez le fichier mots_binaires.cpp et regardez rapidement le code qu'il contient, en particulier, observez la fonction motsbin_classique pour en comprendre le fonctionnement.

(4) Complétez la fonction motsbins_parall : l'algorithme doit être **le même** que celui de motsbin_classique mais les appels récursifs doivent se faire en parallèle avec cilk_spawn.

Appel classique

```
ma_fonction(...)
```

Appel en parallèle

```
cilk_spawn ma_fonction(...)
```

(5) Compilez puis vérifiez que votre implantation fonctionne en lançant le calcul sur plusieurs coeurs de la façon suivante ./mots_binaires −n 8 5. Vous devez obtenir la même ligne de nombre que tout à l'heure.

(6) On va maintenant vérifier que le calcul utilise bien les différents coeurs de la machine. Pour cela, ouvrez un autre terminal et lancez la commande top que vous laisserez tourner. Sur le premier terminal, lancez ./mots_binaires −n 8 30, vous devez alors voir votre programme apparaître dans la liste des applications en cours et le pourcentage du processeur augmenter pour se rapprocher de 800%.

(7) Par ailleurs, si vous lancez ./mots_binaires −n 1 30, le programme apparaît à nouveau mais cette fois le pourcentage du processeur reste en dessous de 100%.

(8) Le calcul en parallèle est-il pour autant plus rapide ? Mesurez le temps de calcul des deux exécutions avec time ./mots_binaires −n 1 30 et time ./mots_binaires −n 8 30. Lancez une fonction avec cilk prend 4 à 5 fois plus de temps qu'un lancement classique ce qui fait que dans ce cas là, comme il y a de nombreux appels de fonctions, le gain de la parallélisation est perdu dans le surcoût. Dans le dernier exercice, nous verrons des stratégies pour contrecarrer en partie cet effet.

Exercice 2 (Instructions processeur)**.**
Dans cet exercice, nous allons explorer les instructions avancées du processeur qui permettent de traiter des vecteurs d'entiers de taille limitée. Nous allons utiliser ces vecteurs pour représenter des permutations de taille maximale 16. Ouvrez le fichier perm_descentes_algo.cpp. Dans la première partie du fichier, nous définissons deux types :

- le type ar16 : c'est un tableau d'entier C++ classique. Nous l'utiliserons pour définir simplement les permutations et pour les opérations classiques sur les tableaux (accès indicés, modification d'une valeur, etc).
- le type perm : c'est un type qui peut contenir 16 entiers de 8 bits et peut être stocké directement sur les registres du processeur. Ainsi, on peut lui appliquer certaines opérations avancées du processeur que nous décrivons par la suite.

Les deux fonctions ar16_perm et perm_ar16 permettent de convertir d'un type à l'autre. Nous en aurons besoin mais il faut en limiter l'usage car il est coûteux en temps (il force une copie en mémoire du tableau).

Le type perm est utile car on peut lui appliquer les opérations suivantes :

- **Permuter les valeurs :** _mm_shuffle_epi8 permet de permuter les valeurs d'une permutation en fonction d'une autre. C'est-à-dire : si j'applique _mm_shuffle_epi8(p1,p2) sur deux permutations, le résultat est une permutation p3 tel que p3[i] = p1[p2[i]].
- **Comparaison termes à termes :** je peux écrire perm p3 = (p1 < p2). Dans ce cas, le vecteur p3 est donné par p3[i] = 255 si p1[i] < p2[i] et 0 sinon.
- **Transformation en nombre binaire :** la fonction _mm_movemask_epi8 transforme un vecteur en nombre binaire : si p[i] > 0 alors le bit de poids i sera mis à 1 et si p[i] == 0 alors le bit de poids i sera mis à 0.

En plus de ces trois fonctionnalités, on utilisera la fonction _mm_popcnt_u32 qui compte le nombre de bits à 1 dans un nombre binaire. Remarquez que bien que ces opérations s'appliquent sur les 16 entiers du vecteurs, elles correspondent à chaque fois à **une seule opération du processeur** ce qui les rend très efficaces. Le but de l'exercice est de remplacer un algorithme classique sur les tableaux (avec boucle for) par une suite bien pensée d'opérations processeurs.

(1) Commencez par compiler le programme avec la commande make perm_descentes_algo et exécutez-le pour observer des exemples d'applications des fonction données ci-dessus. (les permutations sont affichées en hexadécimal)

(2) On veut compter **le nombre de descentes d'une permutation**. On dit qu'une permutation possède *une descente* si p[i] > p[i+1]. La fonction nb_descents_classic vous donne l'algorithme classique pour calculer les descentes d'une permutation. Complétez la fonction nb_descents_optim. Elle doit retourner la même valeur que nb_descents_classic mais : vous **ne devez pas** utiliser de boucle for, vous **ne devez pas** convertir la permutation en type ar16, **vous devez** utiliser uniquement les opérations du processeur décrites ci-dessus. Dé-commentez la fin du programme pour vérifier que votre implantation fonctionne.

Aide : nous avons défini une variable globale decal qui pourrait vous aider.

Exercice 3 (Génération récursive de permutations en parallèle avec calcul des descentes)**.**
À présent, nous mettons en pratique les deux exercices précédents pour calculer à une taille n donnée le nombre de permutations par nombre de descentes. Nous allons compléter le fichier perm_generation.cpp. En lançant le programme ./perm_generation 4, on devra obtenir l'affichage 1 11 11 1 car il existe :

- 1 permutation de taille 4 avec 0 descente
- 11 permutations de taille 4 avec 1 descente
- 11 permutations de taille 4 avec 2 descentes

— 1 permutation de taille 4 avec 3 descentes.

(1) Pour commencer, complétez la fonction affiche_all_permutations. Le but de cette fonction est d'engendrer récursivement les permutations en utilisant les opérations du processeur que nous avons vues dans l'exercice précédent. La fonction ne stocke pas la liste des permutations, elle se contente de les afficher. Elle fonctionnera de cette façon : le paramètre perm p stocke la permutation courante, on affiche la permutation courante uniquement lors de l'arrêt de la récursion pour $n = 1$. Sinon, on effectue n appels récursifs tel que le paramètre passé pour l'appel i soit la permutation p où on a échangé les valeurs p[i] et p[n−1] à l'aide de la fonction permuteij (i va de 0 à $n-1$). Voilà par exemple la pile d'appel pour $n = 3$.

```
Appel initial avec n=3 et p=[0,1,2]
    Appel récursif avec n=2 et p=[2,1,0] (échange pos 0 et 2)
        Appel récursif avec n=1 et p=[1,2,0] (échange pos 0 et 1)
            —> affichage [1,2,0]
        Appel récursif avec n=1 et p=[2,1,0] (échange pos 1 et 1)
            —> affichage [2,1,0]
    Appel récursif avec n=2 et p=[0,2,1] (échange pos 1 et 2)
        Appel récursif avec n=1 et p=[2,0,1] (échange pos 0 et 1)
            —> affichage [2,0,1]
        Appel récursif avec n=1 et p=[0,2,1] (échange pos 1 et 1)
            —> affichage [0,2,1]
    Appel récursif avec n=2 et p =[0,1,2] (échange pos 2 et 2)
        Appel récursif avec n=1 et p=[1,0,2] (échange pos 0 et 1)
            —> affichage [1,0,2]
        Appel récursif avec n=1 et p=[0,1,2] (échange pos 1 et 1)
            —> affichage [0,1,2]
```

Pour tester votre implantation, compilez le programme avec make perm_generation, puis lancez la commande ./perm_generation −a 3 pour afficher les permutations de taille 3. (Remarque, les permutations qui s'afficheront seront toujours de tailles 16 mais les positions supérieures à n n'auront pas été modifiées).

(2) Complétez la fonction nb_descents_optim en reprenant le code écrit pour l'exercice 2.

(3) Complétez la fonction descentes_all_permutations_classique. Cette fonction engendre les permutations de taille n mais, au lieu de les afficher, elle compte le nombre de descentes et incrémente la case correspondant du tableau res (comme on le faisait pour les mots binaires dans l'exercice 1). Votre fonction sera donc similaire à affiche_all_permutations avec uniquement le cas $n == 1$ qui change.

Testez votre implantation en compilant puis en lançant la commande ./perm_generation 4 qui doit afficher

Result : 1 11 11 1 0 0 0 0 0 0 0 0 0 0 0 0

et ./perm_generation 5 qui doit afficher

Result : 1 26 66 26 1 0 0 0 0 0 0 0 0 0 0 0.

(4) Complétez la fonction descentes_all_permutations_parall. C'est la *même chose* que la fonction descentes_all_permutations_classique mais les appels récursifs doivent être lancés avec cilk_spwan pour utiliser le calcul en parallèle.

Testez votre implantation en compilant puis en lançant la nouvelle commande ./perm_generation −n 8 4 qui doit afficher le même résultat que ./perm_generation 4.

(5) Ouvrez un autre terminal et lancez la commande top puis sur le premier terminal, lancez ./perm_generation −n 8 13, vous devez voir apparaître votre programme dans la liste des tâche en cours et l'utilisation du processeur doit se rapprocher de 800%.

(6) Comparez les temps d'exécution grâce aux commandes time ./perm_generation −n 8 13 et time ./perm_generation −n 1 13. Vous remarquez que le passage en parallèle n'améliore pas les performances. En effet, l'appel d'une fonction avec cilk_spawn est 4 à 5 fois plus long que l'appel d'une fonction classique. Ici, nous faisons de la *sur-parallélisation* : nous appelons cilk_spawn sur de trop nombreuses fonctions pour des calculs trop petits. Pour y remédier, on propose de passer en calcul classique pour les « petites valeurs de n ».

(7) Modifiez la fonction descentes_all_permutations_parall pour n'appeler cilk_spawn uniquement si $n > 5$ (sinon, on fera des appels récursifs classiques). Testez à nouveau les performances puis cherchez à modifier cette limite arbitraire de 5 pour optimiser les performances du programme.

(8) Pour aller encore plus vite, on peut dé-recursiver pour les petites valeurs de n...