**TP 3 : Initiation à OpenMP**

## Corrigé et commentaires

### Exercice 1. Définition du nombre de threads

Le nombre de threads peut être défini de plusieurs façons (avec la clause `num_threads`, avec la fonction `omp_set_num_threads` ou la variable d'environnement `OMP_NUM_THREADS`), qui sont testés dans les codes `helloworld1.c`, `helloworld2.c` et `helloworld3.c`, respectivement. Lorsque les trois façons sont utilisées en même temps avec des nombres différents, combien de threads seront effectivement utilisés ?

Voici un petit code `helloworldPriority.c` pour tester la priorité :

```
#include <stdio.h>
#include <omp.h>
int main(){
  omp_set_num_threads(8);
#pragma omp parallel num_threads(16)
  printf("Hello world!\n");
  return 0;
}
```

Compilation et exécution :

```
>> icc -qopenmp helloworldPriority.c
>> export OMP_NUM_THREADS=4
>> ./a.out
```

Résultat de cette expérience : *Hello world !* est affiché 16 fois. Si la clause `num_threads(16)` est retirée, le message sera affiché 8 fois Si l'appel à la fonction `omp_set_num_threads(...)` est également retiré, le message sera affiché 4 fois. On a donc identifié l'ordre de priorité lorsque ces différentes façons sont utilisées en même temps.

La fonction `omp_get_num_threads()` permet d'obtenir le nombre de threads dans la zone parallèle courante. Si la fonction est ajoutée avant le `pragma`, le nombre sera 1, car cette zone est en réalité une zone séquentielle. Si la fonction est ajoutée après le `printf` *(c'est-à-dire, après la région parallèle, car le* ***`pragma`*** *n'agit que sur la commande* ***`printf`****)*, le nombre de threads sera également 1.

### Exercice 2. Région parallèle et portée des variables

Voici le code `variables.c` :

```
#include <stdio.h>
#include <omp.h>
int main(){
  int A = 10;
```

```
5     int B = 20;
6     int C = 30;
7     printf("S: %i %i %i\n", A, B, C);
8
9   #pragma omp parallel default(none) shared(A) private(B) firstprivate(C)
10  {
11    A += omp_get_thread_num();
12    B += omp_get_thread_num();
13    C += omp_get_thread_num();
14    printf("P: %i %i %i\n", A, B, C);
15  }
16    printf("S: %i %i %i\n", A, B, C);
17    return 0;
18  }
```

Compilation, exécution et résultats :

```
>> icc -qopenmp variables.c
>> export OMP_NUM_THREADS=4
>> ./a.out
S: 10 20 30
P: 10 0 30
P: 11 1 31
P: 16 3 33
P: 13 2 32
S: 16 20 30
```

Au début de ce programme, des variables `A`, `B` et `C` sont allouées et initialisées dans une première région séquentielle. Au sein de la zone parallèle, des variables `A`, `B` et `C` sont modifiées. Par défaut, des variables définis avant une zone parallèle seront des variables *partagées* par les différents threads dans la zone parallèle. Cependant, ici, les clauses vont modifier le comportement. Le programme se termine par une nouvelle zone séquentielle, où les valeurs finales des variables sont affichées.

Observations :

- La variable `A` est partagée (clause `shared`). Les 4 threads la modifient en même temps, ce qui est dangereux *(possibilité de 'data race')*. La valeur est 10 au débit *(initialisation avant la région parallèle)*, et 16 à la fin *(accumulation des valeurs 0, 1, 2 et 3)*. Ici, le data race n'est pas visible, mais ça aurait pu être le cas avec un plus grand nombre de threads.
- La variable `B` est privée (clause `private`). La norme ne dit pas comment elle doit être initialisée au début du thread. En fonction des compilateur, le comportement peut être différent. Ici, `icc` initialise la valeur de la variable privée à 0. Ensuite, on voit que chaque thread ajoute sa valeur dans la variable privée, et affiche son numéro de thread. Après la zone parallèle, la variable `B` retrouve sa valeur initiale (20 ici) dans la zone séquentielle.
- La variable `C` est privée avec initialisation (clause `firstprivate`). Même comportement que pour `A`, mais la norme impose que la variable privée soit initialisée avec la valeur de la variable pré-existente. Après la zone parallèle, la variable retrouve sa valeur initiale (30 ici).
- Une variable ne peut pas être listée dans plusieurs clauses.
- Si `default(none)` est utilisé, toutes les variables *pré-existentes* utilisées dans la zone parallèle doivent être listées dans les clauses `shared`, `privite`, etc.

**Exercice 3. Parallélisation de boucles et opération réduction**

Voici la meilleure implémentation du code de diffusion, après vectorisation *(voir corrigé du TP2)* :

```
1  void implementation4(int T, int N, double* C, double* Cnew){
2    double dx = 1./(double)(N-1);
3    double dt = 0.2*dx*dx;
4    double coef1 = dt/(dx*dx);
5    double coef2 = 1 - 4*coef1;
6
7    for(int n=0; n<T; n++){
8      for(int i=1; i<(N-1); i++)
9  #pragma ivdep
10       for(int j=1; j<(N-1); j++)
11         Cnew[N*i+j]
12           = coef2 * C[N*i+j]
13           + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j] + C[N*i+(j+1)] + C[N*i+(j-1)] );
14     switchPointer(&C,&Cnew);
15   }
16 }
```

Pour paralléliser ce code, il faut ajouter la ligne `#pragma omp parallel for` devant l'une des boucles, c'est-à-dire devant les lignes 7, 8 ou 9 du code ci-dessus. Pour l'exercice, j'ai testé les trois alternative dans le code `diffusionIccPar.c`.

Voici les résultats :

- Si le pragma est au dessus de la boucle en $n$, la boucle est effectivement parallélisé *(visible dans le rapport de compilation)*, mais le résultat est faux ! C'est normal, il s'agit d'une boucle 'temporelle' qui n'est pas parallélisable.
- Si le pragma est au dessus de la boucle en $i$ ou en $j$, cette boucle est effectivement parallélisée. Dans le deuxième cas, cela signifie que la boucle en $j$ est parallélisée et vectorisée en même temps.
- Au niveau des temps de calcul *(sur mon MacBook Pro, 4 threads, avec une taille de grille* $1024\times1024$ *et* 1000 *pas de temps)*, j'obtiens 0.89s sans parallélisation, et 0.31s, 0.38s et 1.44s avec les trois parallélisation. La première parallélisation donnant un résultat faux, on l'ignore. Ensuite, c'est la parallélisation de la boucle en $i$ qui donne le meilleur temps. Dans cette expérience, la parallélisation de la boucle en $j$ a ralenti le calcul.

Ceci nous mène à une règle de bonne pratique *(qui n'est pas une règle absolue)* : en principe, on vectorise au niveau le plus bas *(la boucle la plus intérieure)* et on parallélise au niveau le plus élevé *(la boucle parallélisable la plus extérieure)*.

Si vous testez le code sur votre propre machine, avec une taille de grille différente, vous aurez probablement d'autres résultats *(différences de temps plus ou moins marquées)*, mais la conclusion devrait rester la même.

**Exercice 4. Routine dgemm vectorisée et parallélisée**

*Pas de retour sur cet exercice.*