TD n°3

Previously on AMC's OpenMP Videos…

```
#pragma omp atomic {

}

#pragma omp critical <name>
```

Difference entre atomic et critical :

- **Accès atomique** : fondée sur un mécanisme qui existe dans l'assembleur (pas cool :-/). principe du compare and swap : lire une case mémoire, garder la valeur, faire un calcul, relire la même case, si égale a ce qu'on a lu au début, on peux écrire
  Entre la lecture et l'écriture on est sur que personne n'a modifié cette variable.

EXERCICE 1

a)

```
do {
    old _count = count;
    new_count  = old_count + 1;
   success = compareandswap(&count, old_count, new_count);
} while (!success);
```

- Privilégier *atomic* à *critical* : celui-ci est plus léger. (en gros, *critical* sert à rien, juste pour des sections qu'on veut pas faire en double.)

b) La réduction :

```
reduction (operateur : list)
```

Ne fonctionne que sur deux forme de parallélisme :

- `omp parallele`
- `omp for`

version 1

```
int main(int argc, char** argv) {
       int A[N][P], B[N][P];
       int somme = 0;
       float moyenne;
       #pragma omp parallel
       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               #pragma omp atomic
               somme = somme + A[i][j];
           }
       }
       moyenne = somme / (N*P);
       for (int i=0; i <N; i++) {
           for (int j=0; j <N; j++) {
               if (A[i][j] >= moyenne) {
                   B[i][j] = 1;
               } else {
                   B[i][j] = 0;
               }
            }
       }
   }
```

version 2

```
int main(int argc, char** argv) {
       int A[N][P], B[N][P];
       int somme = 0;
       float moyenne;
       #pragma omp parallel private (i,j) shared (A, B, moyenne, somme)
       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               #pragma omp atomic
               somme = somme + A[i][j];
           }
       }
       #pragma omp single
       moyenne = somme / (N*P);
       #pragma omp for
       for (int i=0; i <N; i++) {
           for (int j=0; j <N; j++) {
               if (A[i][j] >= moyenne) {
                   B[i][j] = 1;
               } else {
                   B[i][j] = 0;
               }
            }
       }
   }
```

version 3

```
int main(int argc, char** argv) {
       int A[N][P], B[N][P];
       int somme = 0;
       float moyenne;
       #pragma omp for reduction (+ :somme)
       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               somme = somme + A[i][j];
           }
       }
      //...

   }
```

EXERCICE 2

```
float A[N+2][N+2], diff = 0 ;
int i, j, fini = 0;

int main(int argc, char** argv) {
   read(n);
   initialiser(A);
   calculer(A);
}

void calculer(float **A) {
   int i, j, fini=0;
   float diff=0, temp;
   #pragma omp parallel private(i, j, temp) shared(A, diff, SEUIL, n, fini)
   while (!fini) {
      diff = 0;
      #pragma omp for reduction (+ : diff)
      for (i=0; i<n; i++) {
         for (j=0; j<n; j++) {
            temp = A[i][j];
            A[i][j] = 0.2 * (A[i][j]+ A[i][j-1] + A[i-1][j] + A[i][j+1] + A[i+1][j]);
            diff += abs(A[i][j] - temp);
          }
       }
       if (diff / (n*n) < SEUIL) {
           fini = 1;
       }
    }
}
```

Remarque: En général

- **shared** → si variable déclaré en dehors de la boucle
- **private** → si variable déclaré dans la boucle

Remarque de la remarque : Bon, dans cet exemple, ils ont tout déclarés en dehors, mais l'idée est là :-p Bienvenue dans le monde merveilleux de l'Architecture <3 :/" Welcome to the machine" \o/

EXERCICE 3

```
int max;
init (A);
#pragma omp parallel for
for(i=0; i<N; i++) {
   if (A[i] > max) {
       #pragma omp critical
       if (A[i] > max) {
           max = A[i];
       }
   }

}
```

Exercice 4 :

Question 1

```
int s, i, j;
#pragma omp parallel private(s,i,j)
for(s=0; s<NUM_STEPS; s++) {
   #pragma omp for
   for(i=0; i<N; i++) {
        strength[i] = 0;
        for(j=0; j<N; j++) {
            strength[i] += interaction(i, j);
        }
    }
    #pragma omp for
    for(i=0; i<N; i++) {
        update(i);
    }
}
```

Question 2

Divisons le nombre de calculs :

```
int s, i, j;
#pragma omp parallel private(i,j,tmp)
for(s=0; s<NUM_STEPS; s++) {
    #pragma omp for
    for(i=0; i<N; i++) {
        strength[i] = 0;
    }
    #pragma omp for
    for(i=0; i<N; i++) {
        for (j=0; j < i; j++) {
            temp = interaction(i,j);
            #pragma omp atomic
            strength[i] += temp;
            #pragma omp atomic
            strength[i] -= temp;
        }
        for (i=0; i<N; i++) {
            update(i);
        }
    }
}
```

Question 3

Equilibrons la charge de calculs entre les threads :

```
int s, i, j;
#pragma omp parallel private(i,j,tmp)
for(s=0; s<NUM_STEPS; s++) {
    #pragma omp for
    for(i=0; i<N; i++) {
        strength[i] = 0;
    }
    //
    #pragma omp for schedule (dynamic, 1)
    for(i=0; i<N; i++) {
        for (j=0; j < i; j++) {
            temp = interaction(i,j);
            #pragma omp atomic
            strength[i] += temp;
            #pragma omp atomic
            strength[i] -= temp;
        }
        for (i=0; i<N; i++) {
            update(i);
        }
    }
}
```

Exercice 5

```
int i,j ;
int num_primes = 0;
#pragma omp parallel for private(j)
for (i = 1 ; i<N ;i++){
    bool is_prime = TRUE ;
    for (j=2 ; j<i ; j++){
        if (i%j == 0){
            is_prime = FALSE ;
            break ;
        }
    }
    if (is_prime)
        #pragma omp atomic
        num_primes++ ;
}
```

Résumé: Directives et fonctions OpenMP

- **#pragma omp parallel** : section parallèle
- **#pragma omp parallel for** : boucle **for** parallèle
- **#pragma omp master** : section qui ne sera exécutée que par le thread principal
- **#pragma omp critical** : région "critique" qui ne doit être éxécutée que par un thread à la fois (doit par exemple être utilisé lorsque des threads doivent écrire dans une variable partagée ou dans un fichier)
- **#pragma omp ordered :** section qui doit être exécutée dans l'ordre
- **#pragma omp barrier :** barrière qui permet d'attendre que tous les threads soient arrivés à ce point avant d'éxécuter la suite

On peut ensuite ajouter un certain nombre d'options après les directives (voir les sites cités plus haut pour une description plus complète).

- **private(var1)** : **var1** sera une variable privée de la section parallèle (à noter que la valeur originale déclarée dans la partie non parallèle du programme n'est pas recopiée)
- **shared(var2)** : **var2** sera une variable partagée de la section parallèle (ce qui est le cas par défaut de toute variable qui n'est pas déclarée private)
- **ordered** : la section parallèle comporte un bloc **#pragma omp ordered**
- **schedule(static)** : option par défaut qui réparti les tâches de façon statique entre les threads
- **schedule(dynamic)** : option qui permet de répartir les tâches de façon dynamique, particulièrement utile lorsque les temps d'exécution des tâches sont différents
- **reduction(op:val)** : permet de "réduire" la variable **val** par l'opérateur **op** en sortie de la section parallélisée : par exemple **reduction(+:val)** permet de sommer toutes les variables privées **val** à la fin de la parallélisation.

Enfin, deux fonctions bien utiles :

- **omp_get_num_threads()** : renvoie le nombre de threads qui exécutent une section parallèle
- **omp_get_thread_num()** : renvoie un entier qui correspond au numéro du thread (en partant de 0)