Exercices autour d'OpenMP (suite)

1 Multiplication de matrices

L'objectif est d'étudier différentes versions de parallélisation du produit par bloc de deux matrices carrées de taille N. Pour cela on utilisera le code `block_matmul.c`. On étudie deux approches de parallélisation : le modèle Fork-Join et le modèle à base de tâches.

Approche Fork-Join.

Q1 On parallélise uniquement la première boucle.

Q2 On introduit la clause collapse.

Q3 On utilise un parallélisme emboité pour éclater aussi la deuxième boucle.

Approche en tâches.

Q4 Introduire le parallélisme en tâches.

Comparer les différents temps d'exécution et expliquer les différents comportements.

2 Calcul de pi

On considère le code suivant pour calculer pi

```
int main() {

 double x, pi, sum = 0.0;
 double PI25DT = 3.14159265358979323846264643;
 long num_steps = 100000;
 double step = 1.0/(double) num_steps;
 int start = 1, end   = num_steps;

 for (int i=start;i<= end; i++){
   x = (i-0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
 }

 pi = step * sum;
 printf("pi := %.16e  %.e\n", pi, fabs(pi - PI25DT));
 return 0 ;
 }
```

On parallélise la boucle par quatre différentes approches :

1. On parallélise à la main la réduction. Pour cela, on introduit un tableau `SUM` de dimension le nombre de threads. Chaque thread calcule sa somme locale et la stocke dans le tableau. Puis on évalue la somme globale.

```
double sum[nb_threads];
```

2. Utiliser la directive `atomic` pour mettre à jour la valeur de pi avec la contribution locale.

```
#pragma omp atomic update
```

3. Utiliser la clause `reduction` pour paralléliser le calcul
4. Utiliser la directive `taskloop` avec la clause `reduction` pour paralléliser le calcul

Comparer les temps de calculs en fonction du nombre de threads.
Expliquer ce qui se passe dans la méthode 1. Que faut-il faire pour accélérer cette méthode ?
Reprendre les exemples en utilisant la clause `simd`.