TP OpenMP – Calcul Matriciel

0. **Serveurs (** les **F2** sur le **node26** ; les **F4** sur le **node28** ; vous utiliserez l'**etud** au cas où les nodes ne fonctionnent pas correctement **)**
   - **node26, node28** : CentOS 7.4, Intel Xeon CPU E5-2670 (2 NUMA nodes de 8 cores physiques chacun), gcc 4.8.5, OpenMP 3.1 (201107)

   - **etud** : Fedora release 25, Intel Xeon CPU E5-2650 (2 NUMA nodes de 12 cores physiques chacun), gcc 6.4, OpenMP 4.5 (201511)

1. **Connexion**
   - Connecter vous d'abord sur l'**etud**, puis
   - Connecter vous sur le **node26** ou le **node28** via **ssh**

2. **Prise en main** : éditer le programme `hello.c` suivant

```
#include <stdio.h>
#include "omp.h"

int main()
{

   printf("Before PARALLEL REGION : There are %d threads\n\n" ,
omp_get_num_threads()) ;

   #pragma omp parallel num_threads(4)
   {
      printf("In the PARALLEL REGION : There are %d threads\n\n" ,
omp_get_num_threads()) ;
      printf("Hello World from TID %d!\n", omp_get_thread_num());
   }

   printf("After PARALLEL REGION : There are %d threads\n\n" ,
omp_get_num_threads()) ;
}
```

   - Compiler le programme avec : **gcc -fopenmp hello.c -o hello** ; Vous pouvez ajouter les options habituelles de compilation ; puis l'exécuter
   - Modifier le nombre de threads en utilisant l'une des méthodes suivantes :
     - la clause `num_threads` de `#pragma omp parallel`
     - la variable d'environnement OMP_NUM_THREADS
     - la fonction `void omp_set_num_threads(int num_threads) ;`

3. **Calcul matriciel**

   Soient `A, B` deux matrices carrées d'ordre `n`, on demande d'écrire un programme qui calcule `C=AxB`.

   3.1. Implémenter un programme séquentiel, vérifier les résultats obtenus.

   3.2. Ajout de directive OpenMP compacte « `omp parallel for` » ou « `omp parallel` et `omp for` » à l'initialisation des matrices et au calcul matriciel.

   3.3. Vérifier que le programme parallèle donne des résultats justes en comparant les résultats avec ceux du programme séquentiel, et en faisant varier le nombre de threads et la taille de matrices.

3.4. Vérifier la répartition du calcul entre les threads en utilisant le numéro de thread employé pour chaque itération, ceux pour différent type d'ordonnancement avec `schedule(dynamic/static)`.

3.5. Combien de synchronisations a-t-il dans votre programme parallèle ? Peut-on réduire ce nombre et pourquoi ?

3.6. Mesurer la performance de la parallélization (mesure du temps d'exécution) en variant le nombre de threads et la taille des matrices tout en gardant `dynamic` comme type d'ordonnancement. Interpréter les résultats obtenus.

3.7. Ajouter une boucle itérative sur la multiplication de matrices (ex. $A_{i+1}=A_i \times B$) afin que le volume de calcul soit plus conséquent.

3.8. Mesurer la performance de la parallélization (mesure du temps d'exécution) en variant le nombre de threads et la taille des matrices tout en gardant `dynamic` comme type d'ordonnancement. Interpréter les résultats obtenus.

3.9. Analyser l'effet de l'ajout de la boucle itérative sur la performance du programme. Peut-on l'améliorer ?

4. Utilisation d'une checksum simple pour la comparaison de deux matrices

4.1. Afin de simplifier la vérification des résultats obtenus par les calculs parallèles, nous allons utiliser cette technique pour comparer un résultat obtenu par le calcul séquentiel et un autre par le calcul parallèle.

Pour cela, nous allons calculer un checksum par ligne et par colonne de la matrice `C`. Le calcul de la checksum de la $i^{ème}$ ligne (ou $j^{ème}$ colonne) est : $CSL_i = C_{i,0} + 2*C_{i,1} + \ldots + n*C_{i,n-1}$ (ou $CSC_j = C_{0,j} + 2*C_{1,j} + \ldots + n*C_{n-1,j}$). L'ensemble des checksums forme deux vecteurs : `CSL` et `CSC`. On considère que deux matrices sont identiques si leur `CSL` et `CSC` sont égaux. Si deux matrices ont leurs CSL_i et CSC_j différents, $C_{i,j}$ a une grande chance d'être incorrect.

4.2. Programmer la (les) fonction(s) de calcul des checksums, en séquentiel. Utiliser la (les) pour vérifier les résultats des calculs parallèles précédents.

4.3. Faire une version parallèle de ces fonctions et vérifier qu'elle fournit des résultats justes.

Annexe : Représentation des mesures de performance d'un programme parallèle

La mesure de la performance d'un programme parallèle se fait en mesurant son temps d'exécution en variant le nombre de threads (ou processus) utilisés et la taille du problème traité.

L'ensemble de mesures sera d'abords stocké dans un tableau, puis tracer en courbes, ce qui facilite l'analyse et l'interprétation des résultats.

Exemple :

Tableau 1 : Temps d'exécution du programme parallèle selon la taille du problème et le nombre de threads employés

Taille du problème \ Nombre de threads	1	2	4	8
1000				
2000				
3000				
4000				

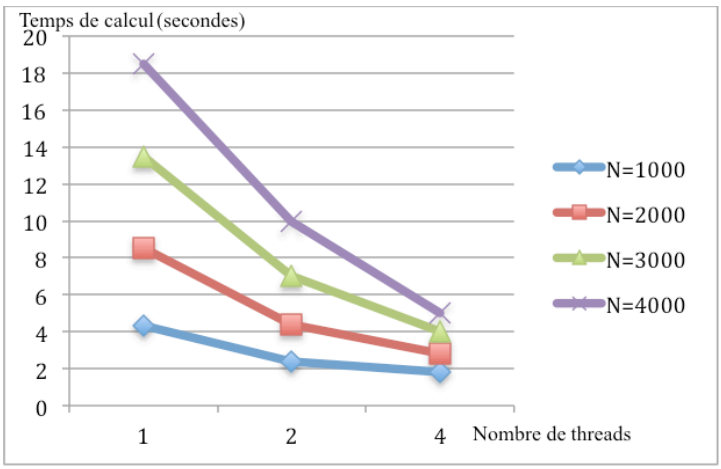

Figure 1 : L'évolution du temps d'exécution du programme parallèle selon la taille du problème et le nombre de threads employés

Attention : le temps d'exécution d'un programme OpenMP contient le temps de calcul, le temps de gestion d'accès à des variables partagées et le temps de gestion des threads parallèles.