TP 1 - Parallélisme et Architecture –

Prise en main OpenMP

(des parties sont rayées car ne peuvent se faire que sur le bladecenter IBM, une machine possédant des processeurs à 8 cœurs).

Pour se TP vous commencerez par travailler sur PC (Linux ou Windows) pour valider le fonctionnement de vos programme. ~~Puis dans un second temps vous vous connecterez par ssh sur une des lames de calcul du Bladecenter : blade01.esiee.fr, blade02.esiee.fr ou blade03.esiee.fr pour y compiler/profiler votre code. Vous utiliserez tous le même login /password pour vous connecter (il sera donné pendant le TP). Il est donc important de créer un sous-répertoire du même nom que votre login esiee afin de ne pas tous travailler dans la racine du compte.~~

`Voir la note n°1 à la fin de ce sujet pour utilisation de windows.`
~~`Voir la note n°2 à la fin de ce sujet pour utilisation du Bladecenter.`~~

~~Il faudra penser **à transférer votre code vers votre compte** à la fin de l'unité.~~

Attention si vous travaillez dans une VM, il faut s'assurer qu'elle soit configurer avec un nombre de cœurs compris entre 2 et le nombre de cœurs physique de la machine réelle.

Exercice 1 – Prise en main

Déterminer le nombre de cœurs de votre architecture, qu'elle est leur fréquence d'horloge : **cat /proc/cpuinfo** sous Linux ou bien le gestionnaire de taches et de périphériques sous windows.

Ecrire un programme C qui se divise en 4 tâches openMP, chacune affichera son numéro de rang. Pour cela il faut :

- Sous linux définir la variable d'environnement OMP_NUM_THREADS : *setenv OMP_NUM_THREADS 4* ou bien utiliser la fonction *omp_set_num_threads(nombre de threads); juste avant le #pragma (Cf. ci-dessous)*
- La directive *#include <omp.h>*
- La directive *#pragma omp parallel* avant la région parallèle qui sera entre accolades,
- La fonction *int omp_get_thread_num ( )* pour obtenir le rang (numéro) d'un thread,
- Si vous compiler sous *icc* il faut utiliser l'option *–openmp*

Remarque : la fonction *int omp_get_num_threads( )* permet d'obtenir le nombre de thread courant (il vaut donc OMP_NUM_THREADS dans une région parallèle).

Exercice 2 – Variables privées

Modifiez (après l'avoir conservé sous un autre nom) l'exercice 1 afin que :

- chacun affiche le contenu d'une variable *VALEUR1* déclarée après le main et initialisée à 1000,
- chacun déclare privée affiche le contenu d'une variable *VALEUR2* déclarée après le main et initialisée à 2000. *VALEUR2* sera déclarée privée à l'entrée de chaque thread à l'aide de la directive *private (VALEUR2)* ajoutée à la fin de la ligne *#pragma omp parallel*. Chaque thread incrémentera la valeur de *VALEUR2*,
- Quelle est la valeur affichée par chaque thread ?
- Modifiez *private* par *firstprivate* et observer le résultat – déduction ?

Exercice 3 – Boucles parallèles

Ecrire un programme qui compte de 1 à 50 en utilisant une simple boucle, et fait un printf du compteur à chaque itération.

Modifiez ce programme pour qu'OpenMP en fasse 2 tâches. Le printf sera modifié pour afficher aussi le rang (numéro) de la tâche qui fait le printf.

Augmenter le nombre de tâches pour qu'il corresponde au nombre de cœurs disponibles.

Exercice 4 – Calcul de PI

Le programme suivant, calcul une approximation de Pi par la méthode des trapèzes. Plus le nombre (nb_pas) est grand plus la précision est grande :

```
#include<stdio.h>
int main () {
      static long nb_pas = 1 000 000 000;
      double pas;
      int i; double x, pi, som = 0.0;
      pas = 1.0/(double) nb_pas;
      for (i=0;i< nb_pas; i++){
            x = (i + 0.5)*pas;
            som = som + 4.0/(1.0+x*x);
      }
      pi = pas * som;
      printf("PI=%f\n",pi);
return 0;
```

1) Commencer par le tester et vérifier qu'il donne le bon résultat !
2) Modifier cette version pour chronométrer sa durée d'exécution. Pour mesurer les durées d'exécution, vous pouvez utiliser la function omp_get_wtime

   Comme la mesure de temps s'effectue sur une machine qui exécute d'autres tâches qui peuvent prendre du temps CPU pour leurs exécutions, il est préférable d'exécuter plusieurs fois ce programme, et de faire la moyenne des durées d'exécutions obtenues. Cette valeur sera la "durée d'exécution monoprocesseur de référence".

3) Transformez le calcul de `Pi` suivant afin de le rendre parallèle avec openMP.

   NB :
   - **Vérifiez toujours le résultat** (en le comparant avec PI de math.h)
   - Augmentez le nombre de pas si la durée de calcul en mono est < ~10s car les résultats ne seront pas significatifs sinon.

4) Exécuter cet algorithme pour 2 threads : commencer par vérifier la valeur obtenue, puis mesurez sa durée d'exécution. N'oubliez pas de faire plusieurs execution et de garder la moyenne.

5) Comparer la durée d'exécution mono-processeur avec la durée 2 threads. A quelle accélération (speed up) doit-on s'attendre ? Quel speedup est obtenu ?

6) Modifier le code pour l'exécuter sur 3, puis autant de threads que de cœurs. Calculez les speedup théorique et mesurer ? Expliquez. **Tracer la courbe avec en abscisse le nombre de threads et en ordonnée la durée d'exécution d'une part, et le speedup d'autre part.**

7) Essayez les différentes stratégies d'ordonnancement (static, dynamic), relevez les performances obtenues.

8) ~~Transférez votre programme sur le bladecenter (Cf. notes), recompilez le et mesurez les durées d'exécutions pour un nombre de threads variant de 1 à 16 threads. Calculez les différents speed up et tracer la courbe avec en abscisse le nombre de threads et en ordonnée la durée d'exécutions d'une part, et le speedup d'autre part.~~

Exercice 4 - Vectorisation

Déclarez 3 tableaux de 20000 float initialisée avec des nombres aléatoires.

Ecrire un programme qui donne : TAB3[i] = TAB1[i] + TAB2[i]*K (où K est une constante quelconque flottante).

Quelle est la version de SSE supportée par votre processeur ? (voir /proc/cpuinfo sous Linux ou le type de votre processeur + recherche google)

Testez :
1. Sans option d'optimisation (attention par défaut, sans argument il y a optimisation, vérifiez dans le man),
2. Cherchez les optimisations (01, 02 et 03) mettant en œuvre SSE, re-tester les.

Vous pouvez vérifier que le code contient de l'assembleur SSE grâce à la commande objdump qui permet de désassembler un binaire et donc voir s'il y a usage de registres et instructions SSE. Observez et commentez le code assembleur généré.

Plutôt que objdump vous pouvez aussi utiliser l'option de compilation –S qui force l'arrêt de la chaîne de compilation avant l'assemblage, il faut alors analyser le fichier assembleur (.s) obtenu. Donnez vos remarques et explications de ce que vous mesurez (il faut vérifier que le compilateur à éventuellement vectorisé automatiquement en utilisant des instructions SSE).

Les résultats seront donnés sous forme de tableau et graphique.

Il faut mesurer les performances avec omp_get_wtime() pour chacune des optimisations et faire un graphique.

Avant / sans optimisation la durée du calcul doit être d'au moins un dizaine de seconde, il faut pour cela une grosse boucle de calcul donc un gros tableau. Mais en agrandissant le tableau il y a risque de "segmentation fault" dû à un dépassement de la taille de la pile si vos tableaux sont déclarés statiquement (<=> int tab[1000000]; dans le main). Deux solutions :

1) Agrandir la taille de la pile lors de la compilation (option de gcc)

2) Allouer le tableau dynamiquement dans le tas (malloc (1000000 * sizeof (in)) ET vérifier que cette allocation a réussi en vérifiant que le pointeur retourné n'est pas égal à NULL.

Attention au phénomène suivant quand vous passez au mode optimisation (-O1, et plus) : si votre programme n'utilise pas les données du tableaux après le calcul, le calcul est purement et simplement supprimé par l'optimiseur de gcc. Vous obtenez alors des valeurs d'exécution très très faible puisqu'aucun code n'est exécuté. Vous vous en apercevrez si vous regarder le code assembleur généré, vous verrez que la boucle n'est plus présente. Vous devez aussi constater l'apparition d'instruction assembleur sse. Pour résoudre cela, il suffit d'afficher une des valeurs du tableau résultat, gcc n'optimisera plus.

Dans les mesures de durée d'exécution, attention à ne pas prendre en compte l'initialisation (random), et les affichages puisque ce sont des appels systèmes non parallélisables.

Notes 1: Sous Windows / Microsoft Visual Studio

Pour créer un projet : new project, puis choisir "Win32 executable application", puis Terminer : vous obtiendrez un projet de base avec une fonction _tmain (et non pas main comme vous aviez l'habitude sous gcc). Pour ce TP choisissez le mode "Release".
Pour activer le support OpenMP il faut cliquez sur le nom du projet, puis properties puis dérouler C/C++ puis dans l'onglet "language" activez le support OpenMP.

Ensuite vous pouvez lancer l'execution depuis Visual ou bien ouvrir un terminal (commande cmd) puis dans votre répertoire lancer votre programme en saisissant son nom.

Notes 2 : Travail sur le Bladecenter

1) Pour vous connecter au Bladecenter à distance
   a. depuis Linux vous pouvez utiliser la commande ssh login@nom_machine:
   b. depuis Windows vous pouvez utiliser les logiciels Putty ou Terraterm qui sont installé

2) Pour transférer votre code
   a. depuis votre PC Linux vers la lame du Bladecenter sous Linux il faut utiliser la commande scp (Secure CoPy). Par exemple la copie d'un fichier d'une machine *serveur1* vers une autre machine *serveur2* se fait avec la commande

```
scp Login1@Serveur1:Chemin1/NomFichier1 Login2@Serveur2:Chemin2/NomFichier2).
```

   b. depuis votre PC windows vers la lame du Bladecenter sous Linux vous pouvez utiliser le logiciel winscp qui a été installé sur vos PCs (et que vous pouvez installer vous même si non disponible car il ne necessite pas de droits particulier).

3) Quand vous êtes connecté à distance sur le serveur, vous pouvez utiliser l'éditeur en mode console « nano » pour éditer votre code. Par exemple « nano mon_programme.c » (control-X pour sauver+quitter etc). L'editeur VIM un peu moins convivial est aussi disponible.

4) Sur les lames de calcul du bladecenter, le chemin d'accès au compilateur n'est pas configuré par défaut, pour l'ajouter il suffit d'exécuter ce script :

En bash : source /opt/intel/bin/compilervars.sh intel64
En csh/tcsh: source /opt/intel/bin/compilervars.csh intel64
Vous disposez alors du compilateur icc et de son man.

Notes 2 : mesure du temps

omp_get_wtime – Elapsed wall clock time

<<Elapsed wall clock time in seconds. The time is measured per thread, no guarantee can be made that two distinct threads measure the same time. Time is measured from some "time in the past", which is an arbitrary time guaranteed not to change during the execution of the program.>>

C/C++:
Prototype: double omp_get_wtime(void);

Pour information uniquement :

Plus généralement, les mesures de temps sous Linux peuvent être effectuées par les fonctions suivantes, qui donnent le nombre de cycles d'horloge. Pour obtenir les temps d'exécution, on exécute les programmes un certain nombre de fois, et on fait la moyenne des temps obtenus en enlevant les valeurs « aberrantes » (très supérieures aux autres).

```
double dtime();
        long long readTSC ();
        long long readTSC () {
        long long t;
        asm volatile (".byte 0x0f,0x31" : "=A" (t));
        return t;
}

double dtime() { return (double) readTSC(); }
```

Mesure du temps d'exécution

```
double t1,t2 ;
t1 = dtime();
//Partie du programme dont on mesure le temps d'exécution.
t2 = dtime();
dt = t2-t1; // Nombre de cycles d'horloge processeur
```

Documentation :
Support de cours
http://www.intel.com/software/products/compilers/clin/docs/main_cls/index.htm