TP - IF5-PAR Parallélisme et Architecture

1ère Partie Parallélisme intra-processeur – OpenMP

Pour se TP vous commencerez par travailler sur PC / Linux, puis vous vous connecterez par ssh sur une des lames de calcul du Bladecenter : blade02 ou blade03 pour y compiler/profiler votre code. Vous utilisez tous le même login/password pour vous connecter, aussi il est important de créer un sous-répertoire du même nom que votre login esiee. Vous transférerez le code de votre PC Linux vers la lame avec la commande scp (pour mémoire 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)`. Il faudra penser à transférer votre code vers votre compte à la fin de l'unité.

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

En csh/tcsh: source /opt/intel/bin/compilervars.csh intel64
En bash : source /opt/intel/bin/compilervars.sh intel64

Vous disposez alors de la commande icc et de son man.

Exercice 1 – Prise en main

Determiner le nombre de cœurs de votre architecture, qu'elle est leur fréquence d'horloge : cat /proc/cpuinfo

Ecrire un programme C qui se divise en 4 tâches openMP, chacune affichera son numéro de rang, puis à l'issue le programme affichera qu'il se termine.

Pour cela il faut :

- Définir la variable d'environnement OMP_NUM_THREADS : *setenv OMP_NU…. 4*
- 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,
- Compiler sous *icc* avec l'option *–openmp*

Remarque : la fonction *int omp_get_num_threads( )* permet de connaître le nombre de thread déclaré par la variable d'environnement ou la dernière exécution de la fonction *omp_set_num_threads( ).*

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 – Conclusion ?

Exercice 3 – Boucles parallèles

Modifiez (après l'avoir conservé sous un autre nom) l'exercice 1 (pas 2 !) :

- ajoutez une variable i après le main, elle servira d'indice de boucle,
- ajoutez for au *#pragma* de l'exercice 1 *: #pragma omp parallel for* (ATTENTION, il doit être placé juste devant la boucle),
- écrire une boucle *for* qui fait varier *i* de 1 à 20,
- chaque thread doit afficher la valeur de i en plus de son rang.

Exercice 4 – Calcul de PI

Transformez le calcul de π suivant afin de le rendre parallèle avec openMP (pour 2 threads). Attention au traitement des variable *som* et *x* : sachant que *#pragma parralel for reduction (+ :som) private (j)* permet de stocker dans *som* la somme des *som* privés de chaque thread, *j* est déclarée privée à chaque thread.

Comparez les durées d'exécution (voir la note en fin de sujet) entre la version mono-thread et la version 2 thread. Pour cela il faut exécuter plusieurs fois l'algorithme en boucle de façon à moyenner les durées d'exécution. Attention, les compilateurs sont capable de supprimer des calculs si leurs résultats ne sont pas utilisés, il faut donc les empêcher de supprimer du code par exemple en utilisant les résultats produits à chaque exécution : vous pouvez les sommer par exemple.

Profilez aussi sur le bladecenter en changeant le nombre de taches utilisées (par exemple de 1 à 16) ?

Remarque : sous linux le fichier */proc/cpuinfo* permet d'obtenir des informations sur le(s) processeur(s).

```
#include<stdio.h>
int main () {
      static long nb_pas = 100000;
      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;
}
```

Pour mesurer les durées d'execution, vous pouvez utiliser la function omp_get_wtime.

omp_get_wtime – Elapsed wall clock time
Description:
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);

Exercice 4 – Pivot de Gauss

Implantez l'algorithme de calcul du pivot de Gauss vu en TP PVM en utilisant OpenMP.

2ème partie - Parallélisme intra-processeur – Cache & SSE

Exercice 2 - Vectorisation

Déclarez 3 tableaux de 20000 float . 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 ?

Testez :

1. Sans option d'optimisation (attention par défaut, sans argument il y a optimisation, vérifiez dans le man), puis avec option « O1 », « 02 », « 03 »
2. Cherchez les optimisations (01, 02 et 03) mettant en œuvre SSE (par exemple : xW, *march=pentium3 -mfpmath=sse -funroll-loops -fomit-frame-pointer sous gcc ?)* re-tester les. Vous pouvez vérifier que le code contient de l'assembleur SSE grace 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 commenterez 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 → regardez le fichier .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.

Exercice 3 – Transposition de matrice

Ecrire un programme C qui effectue la transposition d'une matrice 8 x 8 – Mesurez sa durée d'exécution, compilez sans options, observez et compilez avec l'option qui vous semble la plus adaptée.

Réécrire le code en utilisant les intrinsics (Cf. cours), comparer les durées d'exécution, observez le code assembleur généré. Pour cela vous pourrez utiliser les fonctions __mm__setr_epi…., et __mm_extract_epi…

Notes :

Les mesures de temps sous Linux sont données 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