TP introduction à OpenMP
F. Butelle
10/2015

1 Vérification des capacités du système

Si vous avez un système linux de type debian/ubuntu, on peut recueuillir des informations.
Pour savoir quelle version d'OpenMP est disponible : `dpkg --status libgomp1`
Pour savoir combien de cœurs votre machine dispose : `cat /proc/cpuinfo`
Généralement Linux les numérote linéairement de 0 à $n-1$ (voir les lignes «processor» dans le résultat par exemple avec grep). Cela peut être des cœurs dédoublés vituellement (*hyperthreading*) ; pour avoir le nombre de cœurs physiques, regardez les lignes «cpu cores». On peut aussi consulter la commande `lscpu` si elle est disponible.
1. Écrire un `makefile` avec comme contenu uniquement :

```
CFLAGS= -Wall -fopenmp -O3
```

Cela permettra d'utiliser la commande `make fichier` pour compiler le fichier `fichier.c`.
2. Écrire un programme `exo1.c` avec les directives et fonctions OpenMP nécessaires pour afficher :

- le nbre de processeurs
- le nbre de threads max /limite.
- Votre code doit être compatible avec une exécution sans OpenMP. Indication : testez la valeur de la constante `_OPENMP` par `#ifdef`.

3. Votre programme ne doit pas forcer le nbre de threads. Compilez-le et exécutez-le en essayant de changer la variable d'environnement `OMP_NUM_THREADS`. Vous pourrez pour cela utiliser au niveau *shell*

```
OMP_NUM_THREADS=6; export OMP_NUM_THREADS
```

2 Un cas idéal

On va travailler sur un tableau de taille n. Ce tableau doit être alloué dynamiquement et n doit être passé en premier paramètre du `main`. Le nombre de *threads* utilisé doit être passé en deuxième paramètre (ceci pour pouvoir écrire des scripts Bash facilement en changeant le nombre de *threads*).
Le tableau doit d'abord être initialisé avec des nombres flottants aléatoires (utilisez `rand` et `srand` définis dans `stdlib.h`).
A chaque élément du tableau on appliquera une fonction `f`. Pour chaque fonction vous essaierez avec 2,3,..,16 threads et des tailles de n de 1000, 10^5 , 10^7 et 10^8 . N'hésitez pas à lancer l'exécutable plusieurs fois avec les mêmes paramêtres, il peut y avoir de grandes variations...
1. Ecrire d'abord la version séquentielle avec la mesure du temps de calcul et la paralléliser avec OpenMP. Ne comptez pas le temps d'initialisation du tableau.

On mesurera le temps de calcul avec la fonction `omp_get_wtime()` qui renvoie le temps courant en secondes comme un `double`. Il faut évidemment compiler avec le plus haut niveau d'optimisation.
On utilisera la fonction $f(x) = 2.17 \times x$. Qu'obtenez-vous comme accélération avec 2,3,..,16 threads ? Ne comptez pas le temps de l'initialisation (la fonction `rand` n'est pas *thread safe* et ne doit pas être utilisée en *multithreads*).
2. Essayez avec la fonction $f(x) = 2.17 \times ln(x) \times cos(x)$. Attention c'est la fonction `log` en C et il faut ajouter la librairie `maths` par l'option de compilation `-lm` (le plus simple est de rajouter `LDLIBS=-lm` dans le fichier `makefile`).

3 Somme des éléments d'un tableau

1. Écrire un programme réalisant la somme en parallèle des éléments d'un tableau de taille n (rempli aléatoirement comme dans la section précédente) en utilisant une `réduction`.
2. Mesurer le temps écoulé, faites varier n et le nombre de *threads* pour voir l'évolution. Quel est l'accélération ? N'hésitez pas à lancer l'exécutable plusieurs fois, il peut y avoir de grandes variations...
3. Pour essayer d'améliorer l'accélération, essayez de jouer avec la clause *schedule*, en particulier la taille des blocs/chunks.
4. Et sans utiliser la réduction, mais en rendant atomique l'affectation à la variable d'accumulation, c'est mieux ? Et avec une section critique ?

4 Déboguage de programme OpenMP

Le programme suivant calcule Pi par approximation de l'intégrale de $\frac{1}{1+x^2}$ (qui se trouve être l'arc tangente) entre 0 et 1. En effet, si $f(x) = \frac{4}{1+x^2}$ alors

$$\int_0^1 f(x)dx = 4\int_0^1 \frac{1}{1+x^2}dx = 4(arctan(1) - arctan(0)) = 4(\Pi/4 - 0) = \Pi$$

L'approximation d'une intégrale par une somme de n trapèzes s'écrit :

$$\int_a^b f(x)dx \approx \frac{(b-a)}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{i=0}^{n-1} f(a + i\frac{b-A}{n})\right)$$

Plus n est grand, meilleure est l'approximation.
Le code suivant calcule correctement avec 1 seul thread, mais avec plus d'1 thread le résultat est faux.
Essayez-le et Corrigez-le !
Essayez également avec plusieurs niveaux d'optimisation.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp. h>
#include <sys/time.h>
#define NBITER 400000000
static double f ( double x ) {
  return (4.0/ (1.0 + x*x ) ) ;
}
/* attention code open-MP incorrect ! */
double calcul_pi_parallele (int nb_threads, double a,
                            double b, long  int n ) {
  long int i ;
  double end, start ,xi, valeur_pi = 0.0;
  double somme = 0.0;
  double h = (b-a) / n;

  start = omp_get_wtime ( ) ;
  #pragma omp parallel for num_threads (nb_threads)
  for ( i = 1 ; i < n ; i ++) {
    xi = a+h*i ;
    somme += f(xi) ;
  }

  valeur_pi= h * (somme+(f(a)+f(b)) / 2) ;
  end = omp_get_wtime () ;
  printf ("␣%8.5lf\t␣" , end-start ) ;
  return valeur_pi ;
}

int main (int argc , char ** argv) {
  int nb ;
  double estim =0.0;
  printf ( "Nb␣.␣threads\tTps␣.␣\t\tEstimation␣de␣Pi␣\n" ) ;
  for ( nb=1 ; nb<=omp_get_max_threads ()+2 ; nb++) {
    printf("%d\t\t",nb); fflush(stdout);
    estim=calcul_pi_parallele(nb,0.0,1.0,NBITER);
    printf("%.12lf\n",estim);
  }
  return(EXIT_SUCCESS)
}
```