Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
*** * ***
Université de Carthage
*** * ***
**Institut National des Sciences
Appliquées et de Technologie**

# Compte Rendu N°5

## Calcul de la constante περιφέρεια d'Archimède

Réalisé par :

**Yasmine MHIRI**
**Amal MTIBAA**
**Mohamed Wassim RIAHI**

GL4 – Groupe1

**Année Universitaire : 2018/2019**

## Exercice 1 :

La configuration logicielle et matérielle de la machine utilisée pour cet exercice :

- Processeur : Inter(R) Core(™) i7
- RAM : 6G
- SE : Windows 10 64bits
- IDE : Eclipse java photon

1) Le fichier prog1.c a pour but de connaitre le temps elapsed et le temps cpu, ainsi que l'écart entre la valeur estimée(l'arc cosinus calculé par la fonction trigonométrique acos en indiquant le nombre à traiter qui doit être situé dans un intervalle entre -1 et 1) et la valeur calculée(calculée dans la boucle artificielle en parcourant la matrice pour calculer l'integralle), lors du calcul de la constante d'Archimède par la méthode des rectangles en utilisant la fonction f qui retourne une valeur de type double.
   La formule de cette constante :

$$\int_0^1 \frac{4}{1+x^2}\,dx = \text{constante d'archimède}$$

   ➔ Diviser l'intervalle de l'intégration en sous intervalles puis calculer la somme des aires de ces rectangles.
   La largeur d'un rectangle est égale à h=1/n
   La longueur d'un rectangle est égale à f(h * (i + 0.5)) avec f=4/(1+(h * (i + 0.5))^2)

Après compilation et exécution du programme, on obtient :

```
D:\GL4\archi parall\TP5>gcc -fopenmp -std=c99 -o prog prog1.c

D:\GL4\archi parall\TP5>prog


   Nombre d intervalles       :   30000000
   Valeur calculee      : 0
   |valeur_estime - valeur_calcule | : 3.142E+000
   Temps elapsed              : 1.911E+001 sec.
   Temps CPU                  : 1.911E+001 sec.
```

2) Ajoutons la directive : #pragma omp parallel for schedule(runtime) afin de paralléliser le calcul de la constante

```
  #pragma omp parallel for schedule(runtime)
    // Boucle artificielle a ne pas toucher
for(int k=0; k<100; k++) {


  valeur_calculee = 0;
  for(int i = 0; i<n; i++) {
    double x = h * (i + 0.5);
    valeur_calculee += f(x);
  }
  valeur_calculee = h * valeur_calculee;

}
```

```
D:\GL4\archi parall\TP5>gcc -fopenmp -std=c99 -o prog prog1.c

D:\GL4\archi parall\TP5>prog


   Nombre d intervalles       :   30000000
   Valeur calculee      : 0.192713
   |valeur_estime - valeur_calcule | : 2.949E+000
   Temps elapsed              : 4.999E+001 sec.
   Temps CPU                  : 4.998E+001 sec.
```

On remarque que l'écart type est diminué

Spécifier la politique de partage des itérations : « runtime », Le choix de la politique est reporté au moment de l'exécution.

Si on remplace « runtime » par « static » on obtient :

```
D:\GL4\archi parall\TP5>gcc -fopenmp -std=c99 -o prog prog1.c

D:\GL4\archi parall\TP5>prog


   Nombre d intervalles       :   30000000
   Valeur calculee      : 4.605826
   |valeur_estime - valeur_calcule | : 1.464E+000
   Temps elapsed              : 4.978E+001 sec.
   Temps CPU                  : 4.978E+001 sec.
```

➔ Les itérations sont divisées en blocs de chunk itérations consécutives ; les blocs sont assignés aux threads en round-robin ; si chunk n'est pas précisé, des blocs de tailles similaires sont créés, un par thread.

Si on remplace « runtime » par « dynamic », on obtient :

```
D:\GL4\archi parall\TP5>gcc -fopenmp -std=c99 -o prog prog1.c

D:\GL4\archi parall\TP5>prog


   Nombre d intervalles       :   30000000
   Valeur calculee      : 0.340950
   |valeur_estime - valeur_calcule | : 2.801E+000
   Temps elapsed              : 5.277E+001 sec.
   Temps CPU                  : 5.277E+001 sec.
```

➔ Chaque thread demande un bloc de chunk itérations consécutives dès qu'il n'a pas de travail (le dernier bloc peut être plus petit) ; si chunk n'est pas précisé, il vaut 1.

- On va déplacer la directive dans la boucle intérieure :

```
  // Boucle artificielle a ne pas toucher
for(int k=0; k<100; k++) {
  valeur_calculee = 0;
  #pragma omp parallel for schedule(runtime)
  for(int i = 0; i<n; i++) {
    double x = h * (i + 0.5);
    valeur_calculee += f(x);
  }
  valeur_calculee = h * valeur_calculee;
```

```
D:\GL4\archi parall\TP5>gcc -fopenmp -std=c99 -o prog prog1.c

D:\GL4\archi parall\TP5>prog


   Nombre d intervalles       :   30000000
   Valeur calculee      : -391341328
   |valeur_estime - valeur_calcule | : 3.007E+000
   Temps elapsed              : 2.070E+002 sec.
   Temps CPU                  : 2.070E+002 sec.
```

On remarque que l'écart type est la moitié de celui lorsque la directive est appliquée à la première boucle. Et le temps elapsed et CPU sont supérieurs que l'autre.

3) Ajouter au début du main un test par la directive #ifdef afin d'indiquer à l'utilisateur si le programme est en exécution séquentielle ou parallèle.

```
#ifndef _OPENMP
printf("le programme est en execution sequentielle");
#else
printf("le programme est en execution parallele");
#endif
```

4) La ressource partagée à protéger dans ce programme est la valeur_calculee.
5) OpenMP propose plusieurs solutions pour protéger la modification de variables partagées afin de cumuler des opérations sur des variables.
   - Directive atomic : Permet la mise à jour atomique d'un emplacement mémoire. Cette directive ne prend en charge aucune clause et s'applique sur l'instruction qui vient juste après. Atomic a de meilleures performances que la directive critical.

```
valeur_calculee = 0;
  #pragma omp parallel for schedule(runtime)
for(int i = 0; i<n; i++) {
  double x = h * (i + 0.5);
  #pragma omp atomic
  valeur_calculee += f(x);
}
valeur_calculee = h * valeur_calculee;
```

- Directive critical : S'applique sur un bloc de code, celui-ci ne sera exécuté que par un seul thread à la fois, une fois qu'un thread a terminé un autre thread peut donc avoir accès à cette région critique.

```
valeur_calculee = 0;
  #pragma omp parallel for schedule(runtime)
for(int i = 0; i<n; i++) {
  double x = h * (i + 0.5);
  #pragma omp critical
  valeur_calculee += f(x);
}
valeur_calculee = h * valeur_calculee;
```

6) **Clause reduction() :** Cette clause est utilisée pour calculer une somme, un produit, un maximum... etc. de variables dont la valeur change avec les indices d'une boucle (par exemple) et chaque nouvelle valeur dépend de la valeur précédente.
reduction(operation:variable) :
   - operation : l'opération de réduction utilisée
   - variable : la variable sur quoi l'opération est effectuée

   ➔ atomic et reduction sont plus restrictives mais plus performantes que critical.

```
valeur_calculee = 0;
  #pragma omp parallel for schedule(runtime)
for(int i = 0; i<n; i++) {
  double x = h * (i + 0.5);
  #pragma omp parallel for reduction(+ : valeur_calculee) schedule(runtime)
  valeur_calculee += f(x);
}
valeur_calculee = h * valeur_calculee;
```

7) La clause "reduction" permet de réaliser une opération de réduction sur « valeur_calculee ». Les opérations cumulées sur une variable peuvent être optimisées de manière particulière en utilisant une clause de reduction à la suite de la directive omp parallel for.
La syntaxe générale est reduction(op:variable). La variable stocke les valeurs successives du calcul réalisé par l'opération op. Plusieurs opérations sont possibles :
   - Les opérations arithmétiques (addition +,soustraction -,division /, multiplication *)
   - Les opération binaires (et binaire &, ou binaire |, complément binaire ^)
   - Les opérations booléennes (conjonction &&, disjonction||).

| Opérations arithmétiques | | |
|---|---|---|
| | Fortran | C/C++ |
| sommation | + | + |
| soustraction | - | - |
| produit | * | * |
| division | / | non-utilisé |

| Opérations logiques | | |
|---|---|---|
| | **Fortran** | **C/C++** |
| et | .and. | && |
| ou | .or. | \|\| |
| équivalence | .eqv. | non-utilisé |
| non-équivalence | .neqv. | non-utilisé |

| Fonctions intrinsèques | | |
|---|---|---|
| | **Fortran** | **C/C++** |
| maximum | max | non-utilisé |
| minimum | min | non-utilisé |
| et binaire | iand | & |
| ou inclusif binaire | ior | \| |
| ou exclusif binaire | ieor | ∧ |

- La variable de réduction est valeur_calculee et l'opération est l'addition (+).
  Cette clause est très utile pour les calculs comme les intégrales (comme dans notre exemple), les équations différentielles, …
  Quelle que soit la solution retenue, le résultat est maintenant juste et les performances sont celles attendues.

8) Comparaison des performances :

| **THREADS** | **Temps Elapsed** | | | **Temps CPU** | | |
|---|---|---|---|---|---|---|
| | Atomic | Critical | Reduction | Atomic | Critical | Reduction |
| **1** | 2.662 | 3.250 | 2.421 | 2.662 | 3.250 | 2.421 |
| **2** | 2.651 | 6.855 | 1.693 | 2.651 | 6.855 | 1.693 |
| **4** | 2.720 | 6.025 | 1.365 | 2.720 | 6.025 | 1.365 |
| **6** | 2.827 | | 1.099 | 2.827 | | 1.099 |
| **8** | 3.032 | | 1.036 | 3.032 | | 1.036 |

9) La version avec #pragma omp parallel for reduction est la plus optimale.
   En effet l'augmentation du nombre de thread engendre la diminution du temps .
   Ce résultat est prévu car la clause de réduction est une directive spéciale qui a pour objectif d'améliorer la performance des algorithmes parallèles en enlevant les points de synchronisation, qui demande au compilateur de générer du code qui accumule les valeurs de différentes itérations de boucle d'une certaine manière qui respecte les règles de synchronisation sans avoir à les indiquer explicitement (puisque chacun des threads travaille sur une copie locale et le résultat final sera généré par application de l'opérateur + sur les résultats partiels).