

TP 1 : Préparation

A chaque étape, regarder les messages affichés pour voir si tout s'est bien passé !

1. Récupérer l'archive TP1.tar.gz et extraire les fichiers.
2. Ouvrir un terminal et se placer dans le répertoire I03_TP1 qui vient d'être créé
3. préparer la compilation du code du TP avec les commandes :

```
mkdir -p build  
cd build  
cmake ../src  
cd ..
```

4. Se remettre dans le répertoire I03_TP1 et compiler:

```
make -C build
```

5. Executer le code avec la commande:

./build/PoissonSeq

6. A la fin de l'exécution, les résultats sont sauvegardés au format VTK dans un répertoire "results_..." (le nom précis est affiché à l'écran)

Si on modifie un ou plusieurs fichiers sources (dans le sous-répertoire src), il faut recompiler (point 4).

Si on ajoute un nouveau fichier ou on enlève un fichier existant (dans le sous-répertoire src), il faut adapter les fichiers CMakeLists.txt et refaire les points 3 et 4.

Les commandes ci-dessus génèrent une version optimisée par le compilateur (“Release”). Si nécessaire, on peut compiler une version “Debug” (non optimisée) qui permet d'utiliser un outil de débog (exécution pas à pas, afficher des valeurs en cours de calcul, etc).

Certains des outils donnent aussi plus de renseignements sur une version “Debug” que sur une version “Release”.

Remplacer les commandes de 3. et 4. par:

```
mkdir -p build_debug
cd build_debug
cmake -DCMAKE_BUILD_TYPE=Debug ../src
cd ..
make -C build_debug
```

Mesure du temps de calcul global

Afficher le temps de calcul global avec `time` :

```
time ./build/PoissonSeq
```

A l'écran (exemple) :

real	0m30,283s
user	0m30,186s
sys	0m0,096s

- ▶ Avantage : n'est pas intrusif
pas besoin de modifier le code, ni de le compiler avec des options spécifiques.
- ▶ Désavantage : donne une information globale
on ne sait pas dans quelle partie du code, on passe peu/beaucoup de temps, ni pourquoi.

Mesure plus précise : outil de “profiling” gprof

Fait partie de la famille gcc/g++/gfortran.

- ▶ **Avantage** : calcule le nombre d'appels de chaque fonction et le (pourcentage du) temps qui y est passé
- ▶ **Avantage** : n'est pas très intrusif (pas besoin de modifier le code, mais il faut le recompiler avec une option spécifique: -pg).
- ▶ **Désavantage** : le temps passé dans une fonction est peu précis dans une fonction “courte”
- ▶ **Désavantage** : ne mesure pas le temps dans les différentes parties d'une fonction.
- ▶ **Désavantage** : ne rentre pas dans les bibliothèques dynamiques.

Mode de fonctionnement:

Ajoute dans chaque fonction, un comptage du nombre d'appels de cette fonction et évalue statistiquement le temps passé dans cette fonction (tous les 0.01 secondes on enregistre dans quelle fonction on se trouve).

Utilisation de gprof

Recompiler en utilisant l'option -pg:

```
mkdir -p build_gprof
cd build_gprof
cmake -DCMAKE_CXX_FLAGS="-pg" ../src
cd ..
make -C build_gprof
```

Exécuter le code:

```
./build_gprof/PoissonSeq
```

Collecter les mesures

```
gprof ./build_gprof/PoissonSeq
```

Mesure plus précise : outil de “profiling” perf

Outil spécifique linux

- ▶ **Avantage** : très puissant (mesure le temps passé dans une fonction, une instruction C/C++/fortran, une instruction binaire, multiples indicateurs de performance)
- ▶ **Avantage** : non intrusif
- ▶ **Désavantage** : plus compliqué à utiliser
- ▶ **Désavantage** : ne rentre pas toujours dans les bibliothèques dynamiques.
- ▶ **Désavantage** : nécessite que la machine soit configurée correctement.

Outil à privilégier quand c'est possible

Mode de fonctionnement:

Enregistre les événements dans le noyau Linux, évalue statistiquement le temps passé dans les fonctions et les instructions.

Utilisation simple de perf

Il n'est pas toujours nécessaire de recompiler. Sur certaines machines, une version "Debug" est préférable.

Instrumenter le code (générer les mesures):

```
perf record ./build/PoissonSeq  
(ou perf record ./build_debug/PoissonSeq)  
perf report
```

Voir la documentation de perf pour les (nombreuses) autres options.

Mesure plus précise : outil de “profiling”

valgrind-callgrind

- ▶ **Avantage** : intermédiaire (mesure le temps passé dans une fonction, une instruction C/C++/fortran)
- ▶ **Avantage** : non intrusif
- ▶ **Avantage** : gère mieux les librairies dynamiques.
- ▶ **Avantage** : l'outil kcache-grind offre une interface graphique pratique à utiliser.
- ▶ **Désavantage** : temps d'exécution multiplié par ≈ 40

Mode de fonctionnement:

Le code s'exécute dans une machine virtuelle (simule une machine "idéale"). Permet de mesurer précisément un grand nombre d'indicateurs. Mais explique le facteur de ralentissement.

Utilisation: Utiliser une version "Debug".

Instrumenter le code

```
valgrind --tool=callgrind ./build_debug/PoissonSeq
```

Crée un fichier `callgrind.out.XXXX` où XXXX est le numéro de processus qu'on vient d'exécuter.

Afficher et explorer les résultats de mesure:

```
kcachegrind callgrind.out.XXXX
```

Mesure “manuelle” des temps de calcul.

Principe:

- ▶ encadrer le bloc d'instructions que l'on veut mesurer par des appels à des fonctions qui renvoient la valeur de l'horloge interne de la machine,
- ▶ calculer le temps d'exécution dans ce bloc en faisant la différence des valeurs ci-dessus.

Fonctions système disponibles:

- ▶ `clock()` (fonction système C)
- ▶ `gettimeofday(...)` (fonction système C)
- ▶ `system_clock(...)` (fonction système fortran)
- ▶ `high_resolution_clock` (classe C++ 11)
- ▶ ...

Librairies externes (timers haute résolution) :

- ▶ PAPI : <http://icl.cs.utk.edu/papi>
- ▶ BoostTimers : http://www.boost.org/doc/libs/1_65_1/libs/timer/doc/cpu_timers.html

Remarques

Chaque fonction système est utilisable ou non suivant le langage de programmation utilisé.

La précision n'est pas la même: consulter la documentation.

Exemple avec la fonction C système clock:

```
#include <stdio.h>
#include <math.h>
#include <time.h>
int main()
{
    clock_t t1, t2;
    t1 = clock();

    int n=100000;
    double s = 0.0;
    for(int i = 0; i < n; i++) s += sin((0.2*i)/n);

    t2 = clock();
    float diff = (float)(t2 - t1)/CLOCKS_PER_SEC;
    printf("temps calcul : %f s", diff);
    return 0;
}
```

On fournit une classe C++ Timer avec les fonctionnalités suivantes:

```
//definit une variable "chronometre"  
Timer T;  
//demarrer le chronometre T  
T.start();  
//arreter le chronometre T  
T.stop();  
//remettre a zero  
T.reset();  
  
// retourne le temps mesure  
// entre un appel start() et un stop()  
// (utiliser juste apres le stop())  
double dt = T.elapsed();
```

Se mettre dans le répertoire I03_TP1, faire une copie du répertoire src dans Q1/src et compiler:

```
mkdir -p Q1/build  
cp -rf src Q1/src  
cd Q1/build && cmake ../src && cd ..  
make -C build
```

Q1. Utiliser la classe Timer pour mesurer séparément

- ▶ le temps d'initialisation (lignes 31-32 du fichier src/main.cxx)
- ▶ la moyenne du temps d'exécution d'une itération de calcul (lignes 43-45 du fichier src/main.cxx)

Executer le code avec et sans l'option out=10:

```
./build/PoissonSeq  
./build/PoissonSeq out=10
```

(la seconde exécution sauvegarde les résultats sur fichier)

Se mettre dans le répertoire I03_TP1, faire une copie du répertoire Q1/src dans Q2/src et compiler:

```
mkdir -p Q2/build
cp -rf Q1/src Q2/src
cd Q2/build && cmake ../src && cd ..
make -C build
```

Q2. La classe `Values` représente un ensemble de $n_0 \times n_1 \times n_2$ valeurs et possède des fonctions (`operator()`) qui retournent la composante $v_{i,j,k}$ où v est de type `Values`.

- ▶ Décrire comment sont rangées en mémoire les composantes $v_{i,j,k}$ (examiner les fichiers `src/values.cxx` et `src/values.hxx`).
- ▶ Changer la façon de ranger les composantes en mémoire (il y a plusieurs possibilités)
- ▶ Comparer les résultats et les temps de calcul avec la version dans Q1.
- ▶ Expliquer les différences éventuelles de temps calcul (examiner aussi le fichier `scheme.cxx`).