TP 1. Optimisation séquentielle

Conseil

Dans les séances du cours I03 du master AMS, on fournira des fichiers à utiliser comme support de cours ou de TP.
On vous conseille de créer un répertoire vide dans votre espace de travail où vous copierez ces fichiers et où vous travaillerez dans le cadre de ce cours.
Ceci afin d'éviter de mélanger les fichiers de ce cours avec ceux utilisés lors d'autres enseignements.

Préparation

Récupérer l'archive compressée `TP1.tar.gz` et extraire les fichiers qui sont contenus dans cette archive :

```
cd <repertoire dans votre espace de travail>
cp /home/t/tajchman/AMSI03/2018-11-30/TP1.tar.gz .
tar xvfz TP1.tar.gz
```

Se placer dans le répertoire `TP1` :

```
cd TP1
```

et compiler avec la commande ci-dessous :

```
./build.sh
```

Remarque

`build.sh` est un fichier de commandes unix (dans le répertoire `TP1`) qui contient les commandes pour compiler les codes dans plusieurs configurations. Les différentes configurations sont :

- une version "release", optimisée par le compilateur (répertoire `install/Release`),
- une version "debug" permettant le suivi de l'exécution par un "debugger" et qui donne plus d'informations avec les outils présentés plus loin (répertoire `install/Debug`),
- une version "profile" : "debug" + comptage des appels de fonctions + échantillonage en cours d'exécution (répertoire `install/Profile`).

Règle suggérée :

- Pendant la mise au point du code : utiliser la version debug.
- Pour (essayer d') améliorer le temps d'exécution : utiliser la version profile ou la version debug + outil de mesure du temps
- Quand on est satisfait de l'état du code : utiliser la version optimisée.

1 Outils de mesure du temps calcul

Il existe de nombreux moyens de mesurer le temps d'exécution de code ou de parties de code :

- Commande unix `time` : mesure globale (temps ressenti par l'utilisateur)
- Fonctions définies par le langage et utilisables depuis l'intérieur du code :
  - `second(...)` (fortran),
  - `gettimeofday(...)` (C/C++),
  - `std::clock` (C++),
  - `tic/toc` (matlab),
  - ...

  Permet de mesurer le temps d'exécution d'un groupe d'instructions.
  Penser à vérifier dans la documentation quelle est la précision des mesures.
- Librairies, par exemple PAPI
  (`https://icl.cs.utk.edu/projects/papi/wiki/Main_Page`)
  Permet de consulter des compteurs système très bas niveau (par exemple : nombre d'opérations flottantes, utilisation des caches, utilisation des registres, etc.)
- Outils externes de "profilage", ajoutent automatiquement des points de mesure dans le code (gprof), s'interposent entre le code et le système pour récuperer des informations (valgrind, perf)

  exemples : `gprof`, `perf`, `callgrind (valgrind)` (outils sous unix/linux), `vtune` (intel), etc.

  Permet de connaître des informations intermédiaires : nombre d'appels et temps moyen d'exécution de fonctions par exemple.

Les outils de mesure perturbent les temps de calcul et, en général, il faut les utiliser avec une version debug. Ils donnent seulement une indication sur les endroits du code les plus intéressants à optimiser. De toute façon, à la fin, il faut mesurer les temps calculs sur la version "release" (on a parfois des surprises) .

1.1 Appels explicites de fonctions système depuis le code source

Les fichiers source `src/util/timer.cpp` et `src/util/timer.hpp` contiennent une classe C++ qui utilise la fonction standard C++11

```
std::chrono::high_resolution_clock::now
```

pour mesurer le temps entre 2 positions dans un code.

Cette classe est utilisée dans le fichier source `src/valeur_propre/power1.cpp` qui initialise une matrice 100×100 et calcule sa plus grande valeur propre (en valeur absolue), par la méthode de la puissance :

$$\lim_{k \to \infty} ||A^k v||/||A^{k-1} v||.$$

Question 1.

Examiner le fichier `src/valeur_propre/power1.cpp` et la classe `Timer` (fichiers `src/util/timer.cpp` et `src/util/timer.hpp`).

Exécutez le code `install/Release/power1` (compilé au début du TP). Le code affiche, entre autres, le temps d'initialisation de la matrice, le temps de calcul de la valeur propre et le temps total de calcul.

Ré-exécutez le code plusieurs fois.

Les temps mesurés varient légèrement. Expliquer cette variation.

1.2 Mesure globale du temps calcul

Question 2.

Utiliser la commande `time` pour afficher le temps de calcul total ressenti par l'utilisateur (temps entre le moment où l'utilisateur appuie sur la touche "Entrée" du clavier et le moment où l'exécution se termine) :

```
time install/Release/power1
```

Le temps affiché par `time` est normalement (un peu) supérieur au temps total affiché par le code (voir Question 1.). Expliquez la différence.

Conclusions : quand on mesure des temps calcul, la machine doit exécuter le moins possible de tâches non liées au code, et il faut faire une étude statistique sur plusieurs exécutions.

1.3 Utilisation d'un outil de "profilage"

Ce type d'outil insère automatiquement pendant la compilation des instructions à chaque entrées et sorties dans des fonctions du code source. On dit aussi qu'on instrumente le code source.
On utilisera ici un outil standard `gprof`, disponible avec la suite de compilateurs `gcc/g++/gfortran`.

Question 3.

Au début du TP, la commande build.sh a généré une version "profile" (répertoire `install/Profile`), utilisable avec `gprof`.

Pour information, cela consiste à utiliser les options du compilateur "-g" (compilation en mode debug) et "-pg" (compilation en mode "profile").

Exécuter le code en version "profile" :

```
install/Profile/power1
```

L'exécution produit un fichier nommé `gmon.out`. Les informations contenues dans ce fichier doivent être retraitées par la commande

```
gprof install/Profile/power1 >& res.gprof
```

Examiner le contenu du fichier `res.gprof`

1.4 Utilisation de l'outil `valgrind`.

`valgrind` exécute les codes dans un environnement contrôlé (machine, processeur, mémoire virtuelles) où tous les appels système, les accès à la mémoire, etc., sont examinés.
Dans ce mode de fonctionnement, les temps de calcul sont beaucoup plus grands, mais néanmoins significatifs quand on compare les temps de calcul de différentes parties et/ou versions du code.
Remarque

On utilise souvent `valgrind` pour vérifier l'utilisation correcte de la mémoire.

Question 4.

Exécuter le code en version "debug" sous contrôle de `valgrind` :

```
valgrind --tool=callgrind install/Debug/power1
```

L'outil produit des informations dans un fichier `callgrind.out.XXX` (chaque exécution de `valgrind` génère un fichier de nom différent).

Il est possible d'examiner le contenu de ce fichier, mais `valgrind` fournit un utilitaire pratique :

```
kcachegrind callgrind.out.XXX
```

où il faut remplacer `callgrind.out.XXX` par le nom exact du fichier produit.

Utilisez cet outil pour explorer les mesures de l'exécution du code.

1.5 Utilisation de l'outil perf

`perf`, un outil disponible sous linux peut être très intéressant, en particulier pour des études de très bas niveau (mesure précise des accès mémoire, profilage du langage machine, compteurs bas niveau, etc.). L'outil utilise les fonctionnalités du noyau Linux pour mesurer les événements (compteurs internes du systèmes). Son utilisation efficace requiert une certaine expertise.
Il n'est pas disponible sur les machines utilisées pendant les TP, mais vous êtes encouragés à le tester s'il est installé sur d'autres machines.
Il s'utilise sur des codes compilés en mode debug en 2 étapes :
Exécution du code sous contrôle de perf

```
perf record -e instructions <code>
```

(il existe d'autres options que celle utilisée ici)
Examen du rapport de perf

```
perf report
```

(permet de naviguer en mode texte dans les résultats, des options de perf report produisent des documents texte, html, pdf ou autres).

2 Techniques d'optimisation séquentielle

On utilise ici un code qui calcule (une approximation de) la plus grande valeur propre d'une matrice, dans le fichier source `src/valeur_propre/power1.cpp` (voir le 1.1 page 2). L'essentiel du temps calcul est passé dans la fonction `produit_matrice_vecteur`.

2.1 Tentative d'optimisation 1

Question 5.

Exécuter les codes `install/Release/power1` et `install/Release/power2`, comparer les temps de calcul.

Expliquer les différences de temps calcul en examinant les fichiers sources C++ `src/valeur_propre/power1.cpp` (utilisé dans le code power1) et `src/valeur_propre/power2.cpp` (utilisé dans le code power2).

Question 6.

Faire la même comparaison avec les codes `install/Release/power1f` et `install/Release/power2f`, qui utilisent respectivement des sources fortran `src/valeur_propre/power1.f90` et `src/valeur_propre/power2.f90`.

Expliquer les différences de résultats entre la version C++ (`power1.cpp` et `power2.cpp`) et fortran 90 (`power1.f90` et `power2.f90`).

2.2 Tentative d'optimisation 2

Question 7.

Exécuter les codes `install/Release/power2` et `install/Release/power3`, comparer les temps de calcul.

Expliquer les différences éventuelles de temps calcul en examinant les fichiers sources C++ `src/valeur_propre/power2.cpp` (utilisé dans le code power2) et `src/valeur_propre/power3.cpp` (utilisé dans le code power3).

2.3 Tentative d'optimisation 3

Question 8.

Exécuter les codes `install/Release/power3` et `install/Release/power4`, comparer les temps de calcul.

Expliquer les différences éventuelles de temps calcul en examinant les fichiers sources C++ `src/valeur_propre/power3.cpp` (utilisé dans le code power3) et `src/valeur_propre/power4.cpp` (utilisé dans le code power4).

3 Transposition de matrice

3.1 Parcours par lignes ou par colonnes

On s'intéresse ici à l'opération de transposition des matrices :

$$A^T = (a^T_{i,j})_{i=1,\dots,n,j=1,\dots,n} = (a_{j,i})_{i=1,\dots,n,j=1,\dots,n}$$

où $a_{i,j}$ est le coefficient de la matrice d'origine à la ligne i et la colonne j.

Question 9.

Exécuter les codes `install/Release/transpose1` et `install/Release/transpose2`.

Comparer les temps de calcul et expliquer les différences en examinant les fichiers source `src/transposee/transpose1.cpp` et

`src/transposee/transpose2.cpp`.

3.2 Algorithme par bloc - version 1

On garde la structure des matrices comme dans `transpose1.cpp` et `transpose2.cpp`. Par contre le parcours de indices de matrice est différent.

Question 10.

Exécuter le code `install/Release/transpose3`.

Comparer les temps de calcul avec les 2 versions précédentes et expliquer les différences en examinant le fichier source `src/transposee/transpose3.cpp`.

3.3 Algorithme par bloc - version 2

Dans cette version, on utilise une structure des matrices par bloc. Chaque bloc est lui-même une matrice à coefficients scalaires. L'algorithme s'écrit formellement de la même façon.

Question 11.

Exécuter le code `install/Release/transpose4`.

Comparer les temps de calcul avec la version précédente et expliquer les différences en examinant le fichier source `src/transposee/transpose4.cpp`.