TP3 : Performance énergétique d'un code

Objectif

Dans le deuxième TP, nous avons surtout étudié l'impact de la fabrication d'un ordinateur. En effet, la partie usage était très simplifiée. L'objectif de ce TP est d'étudier les performances énergétiques d'un code. Ce TP est fortement inspiré d'un TP proposé par Gaël Guennebaud à des étudiants et étudiantes de licence d'informatique de l'université de Bordeaux.

Ce TP a une durée de 4 h environ. Un compte-rendu de 4 pages (2 feuilles recto/verso) est attendu. Il faudra déposer ce rapport sur Moodle, en PDF.

Dans ce TP, nous allons explorer quelques leviers d'actions pour rendre un code de calcul plus efficace énergétiquement. Il s'agit d'un sujet extrêmement vaste et dans le cadre de ce TP, nous en explorerons seulement quelques aspects :

- Langage : Python vs Python compilé (à la volée) vs C++
- Parallélisme : code séquentiel vs SIMD (Single Instruction, Multiple Data) vs multithreading vs pipelining

Pour cela nous prendrons comme cas d'étude le calcul de l'ensemble de Mandelbrot (lien) qui, en quelques lignes de code, permet de générer des images captivantes.

Remarques préliminaires

Choix de l'ordinateur Pour ce TP, vous avez deux possibilités :

- soit utiliser votre propre ordinateur et dans ce cas les installations à faire au préalable sont données en annexe ;
- soit utiliser l'une des machines de l'école.

Ne prenez la première option que si vous connaissez bien votre ordinateur et que vous êtes à l'aise avec l'installation de bibliothèques tierces.

Téléchargement du code Télécharger et décompresser l'archive suivante : Code.

Test préalable avec s-tui Ouvrir un terminal et lancer *s-tui* :

```
s-tui
```

La puissance qui vous intéresse est la puissance globale : "Power - package-0,0". Prenez une valeur de *Refresh* assez petite (par exemple 0,2 s). Pour sauvegarder les valeurs dans un fichier :

```
s-tui --csv-file logfile_scenario_01.csv
```

Le fichier *logfile_scenario_01.csv* contient alors toutes les informations de mesure et le relevé s'arrête lorsque l'on arrête *s-tui* (conseil : travaillez avec un fichier CSV par scénario étudié). Les mesures sont aussi affichées dans l'interface du terminal. Vous pouvez par exemple déjà repérer la puissance au repos (P_{idle}). N'hésitez pas à réduire la taille de la fenêtre du terminal pour éviter d'avoir une puissance au repos trop importante.

Contact actuel au sujet des TP : Adrien Vincent – adrien.vincent@bordeaux-inp.fr
Autrice originelle : Annabelle Collin

Contexte de travail *Si vous êtes sur votre propre machine, il faut que vous branchiez votre ordinateur sur le secteur.* Ensuite, utilisez un éditeur de texte qui ne biaise pas significativement vos mesures en lançant des tâches coûteuses : par exemple *gedit* (voir l'impact sur la puissance au repos P_{idle}). Pour noter les mesures ou faire des recherches internet, utilisez (si possible) un autre ordinateur afin de ne pas perturber vos **mesures de consommation**.

Utilisation de *s-tui* (machines persos sous Linux ou ordinateur de l'école) En cas d'utilisation de *s-tui* ou de relevé de puissance variant sensiblement au cours d'une tâche de calcul, un script d'analyse ad-hoc est à disposition ici. Commencez par consulter le document *Quick Start Guide* pour connaître la procédure d'utilisation typique de ce script.

Python

1. Nous allons commencer les mesures avec le script python *mandelbrot.py* :

```
python3 mandelbrot.py
```

Ce script affiche le temps de calcul de l'image (dépendant de la taille de cette dernière qui se fixe avec le paramètre n), puis la stocke dans le fichier mandelbrot.jpg. Le temps d'écriture de l'image n'est pas comptabilisé car 1) nous n'avons pas la main sur cette étape, et 2) l'énergie absorbée lors de cette étape n'est pas homogène avec celle du calcul. Ce premier « run » est assez long, c'est normal. Après avoir noté la puissance active, profitez-en pour préparer un document dans un tableur, tel que *LibreOffice* par exemple, dans lequel vous noterez toutes vos mesures. Celui-ci doit posséder un certain nombre de colonnes, telles que :

1. l'identifiant de l'expérience, par exemple « 01 » ;
2. une description de l'expérience, par exemple « Python, n = 1048, numba+parallel » ;
3. la durée d'exécution, en secondes ;
4. la puissance active moyenne observée pendant l'exécution de la tâche de calcul, en watts ;
5. la puissance moyenne au repos, consommée en arrière-plan, en watts ;
6. une formule calculant l'énergie absorbée par la tâche de calcul, en milliwattheures ;
7. une formule calculant les émissions de CO_2.

Après exécution, vérifiez qu'une image "mandelbrot.jpg" se trouve bien dans le dossier et n'oubliez pas de reporter le temps d'exécution !
Remarque : ressource intéressante pour les émissions de CO_2 : Carte électricité.

2. Pour la deuxième expérimentation, nous allons tester la compilation à la volée de la fonction mandelbrot via *numba*, qui est un compilateur JIT (« Just In Time »). Pour cela, il vaut faut ajouter dans le code *mandelbrot.py* :

1. l'import `from numba import jit, prange`
2. et le décorateur `@jit(nopython=True)` juste avant la définition de la fonction mandelbrot.

Lancez une mesure et comparez la consommation énergétique.

À partir de maintenant, vous allez faire les calculs pour plusieurs tailles d'images allant de 256×256 à 2048×2048 (en remplaçant la ligne n=168 de la fonction *main*). Conservez bien l'ensemble de vos résultats.

3. La version précédente est nettement plus rapide et moins énergivore, mais elle n'exploite qu'un seul des cœurs de votre CPU.
Trouvez combien de cœurs physiques possède votre CPU en faisant une recherche sur internet à partir du modèle.

Remarquez que chaque pixel de l'image peut être calculé indépendamment des autres. Nous pouvons donc faire calculer chaque ligne de l'image par un cœur différent, en nous appuyant sur Numba pour faire ce travail :

1. Ajoutez l'option parallel=True au décorateur @jit : `@jit(nopython=True,parallel=True)`
2. Demandez explicitement à Numba de paralléliser la boucle sur les lignes en remplaçant `range` par `prange` (ici le préfixe « p » est pour parallèle)

Relancez une mesure. Le calcul est effectivement plus rapide sur plusieurs cœurs, mais la puissance absorbée est-elle plus importante que précédemment ? Enfin, qu'en est-il du côté de la consommation énergétique ?

Point d'étape : appelez votre chargé·e de TP

C++

4. Peut-on faire mieux avec un langage compilé (de manière anticipée, *Ahead-Of-Time*) comme C++ ? Pour vérifier cela, nous allons commencer par une version équivalente au code Python et compilé sans parallélisation. Tout d'abord compilez :

```
clang++ -I eigen -O3 mandelbrot.cpp -o mandelbrot
```

et ensuite exécutez (vous pouvez mettre en argument la taille de l'image pour ne pas recompiler pour toutes les tailles choisies) :

```
./mandelbrot 2048
```

Comme *Numba*, le compilateur *clang* est basé sur *LLVM* pour la génération du code. Comparez les temps de calcul et la consommation énergétique avec la version équivalente de *Python+Numba*. Vous noterez également le nombre moyen d'itérations effectuées par pixel dans une colonne.

5. Pour tester le multi-threading :
   1. Ajoutez la directive `#pragma omp parallel for` avant la boucle à paralléliser : ici la boucle sur les colonnes `for(int i = 0; i < n; ++i)`.
   2. Ajoutez l'option de compilation *-fopenmp* à la ligne de commande.

      [Linux / Windows]

      ```
      clang++ -I eigen -O3 -fopenmp mandelbrot.cpp -o mandelbrot
      ```

      [MacOS]

      ```
      clang++ -I . -I eigen -std=c++11 -Xclang -fopenmp -lomp -O3 mandelbrot.cpp -o mandelbrot
      ```

      ou si les bibliothèques installées par *Homebrew* ne sont pas incluses automatiquement :

      ```
      clang++ -I/opt/homebrew/include -L/opt/homebrew/lib -I . -I eigen -std=c++11 -Xclang -fopenmp -lomp -O3 mandelbrot.cpp -o mandelbrot
      ```

Faites une mesure et comparez à la mesure précédente.
Astuce : pour faciliter la lecture de la puissance absorbée, vous pouvez allonger la durée d'exécution en recalculant plusieurs fois la même image, par exemple 4 fois avec la commande :

```
./mandelbrot 2048 4
```

6. Cette version sous-exploite encore grandement les capacités de calcul de nos CPU dont chacun des cœurs est capable de réaliser une même opération sur plusieurs données en même temps. Ce type de parallélisation s'appelle SIMD (*Single Instruction Multiple Data*). Par exemple, vos CPU sont capables de réaliser 4 opérations flottantes double précision en même temps via

le jeu d'instruction AVX. On parle de jeux d'instructions « vectorielles ». Dans certains cas, le compilateur est capable de « vectoriser » automatiquement certaines boucles de calcul. Ce n'est pas le cas ici. Nous avons donc dû écrire une version spécifique de la fonction *mandelbrot* calculant 2, 4, 8 ou plus de pixels en même temps. Il s'agit de la fonction *mandelbrot_simd* du fichier du même nom. Testez cette version. Il faudra en plus ajouter les options *-mavx -mfma* au compilateur pour exploiter au mieux vos CPU. Faites deux mesures de cette version : une sans multithreading (sans *-fopenmp*) et une avec.

7. Le pipelining est un parallélisme au niveau des instructions inspiré du travail à la chaîne. Sur les CPU x86-64 actuels, une opération élémentaire nécessite 4 cycles pour être exécutée. S'il y a suffisamment d'instructions indépendantes à traiter, ces CPU sont capables de les traiter à la chaîne de telle sorte qu'à chaque cycle une instruction démarre et une autre se termine. Dans ce but, la version précédente de *mandelbrot* est capable de traiter plusieurs colonnes de l'image en même temps, par exemple 4. Pour cela, il vous faut remplacer le 1 par 4 dans l'appel de la fonction *mandelbrot_simd* :

```
count = mandelbrot_simd<double,4>(img, ...
```

à la ligne 121. Faites deux mesures de cette version : une sans multithreading (sans *-fopenmp*) et une avec (les deux mesures seront faites avec *-mavx -mfma*).

Point d'étape : appelez votre chargé·e de TP

Comparaison

8. Il est temps de tirer quelques conclusions. Du point de vue de la consommation énergétique :
a. Quelle est la meilleure version ?
b. Une version plus rapide qu'une autre est-elle toujours clairement la meilleure ? Ou, autrement dit, est-il suffisant d'optimiser le temps d'exécution pour optimiser la consommation énergétique ?
c. Entre optimiser l'utilisation d'un seul cœur (SIMD+pipelining) ou simplement utiliser le multithreading seul, quelle option est la plus intéressante ?
[Bonus] Pourquoi, avec les versions SIMD, le nombre moyen d'itérations par pixel est-il plus élevé ?

9. Dans les expérimentations précédentes nous n'avons pas pris en compte le **coût de la compilation de code** C++. Dans un contexte où le binaire généré est utilisé de très nombreuse fois pour explorer les méandres de l'ensemble de Mandelbrot, ce coût est négligeable. Mais à partir de combien d'images calculées une solution C++ (par exemple la moins optimisée) est-elle moins gourmande en énergie que la meilleure solution Python ? À vous de répondre à cette question en faisant la ou les mesures adéquates.
Astuce : pour mesurer le temps de compilation vous pouvez utiliser la commande *time* :

```
time clang++ ...
```

Point d'étape : appelez votre chargé·e de TP

10. Que se passe t-il si l'on change le pays dans lequel on effectue la tâche de calcul ? Faire une étude en sélectionnant un pays avec un mix électrique plus faiblement carboné que la France, ou au contraire plus carboné.

11. Afin de trouver le meilleur compromis énergétique, quel(s) autre(s) paramètre(s) serait-il intéressant d'étudier ? Comment pourrait-on les intégrer ?

12. Mise en application :

- Sélectionner un exemple numérique de votre formation que vous présenterez.
- Évaluer la consommation énergétique et les émissions CO_2 associées.
- Proposer une (ou des) alternative(s) pour en réduire la consommation énergétique.

Annexe : Installation préliminaire pour faire les mesures sur votre ordinateur personnel

Mettez-vous bien sur l'OS principal de votre machine (pas de machine virtuelle telle que *VirtualBox*). Vous devez être administrateur/administratrice de votre ordinateur afin de pouvoir accéder au suivi de la consommation électrique (s'agissant d'un vecteur d'attaques informatiques assez classique).
FAIRE LES ÉTAPES DANS L'ORDRE –
Prenez votre temps pour ne pas en perdre après ! ...

Première étape - Préparation de l'OS
[Linux] - Passer à l'étape suivante.

[Windows] - Installer WSL (*Windows Subsystem for Linux*) qui est une couche de compatibilité permettant d'exécuter des exécutables binaires Linux (au format ELF) de manière native sur Windows 10 et Windows Server 2019.

1. Ouvrir PowerShell sous Windows 10/11 en tant qu'**administrateur**
2. Lancer la commande

```
wsl --install
```

   Si cela ne fonctionne pas aller dans le Store de **Windows** et récupérer directement *WSL Ubuntu*.
3. Redémarrer votre ordinateur
4. Deux possibilités : soit le terminal Ubuntu s'ouvre tout seul ; soit vous devez aller le chercher dans les applications en tapant *Ubuntu*
5. Créer un compte sous ce Terminal Ubuntu
6. Le disque C se trouve sous `/mnt/c` donc pour avoir accès au dossier qui contient le code, il faudra faire quelque chose comme :

```
cd /mnt/c/Documents And Settings/[...]
```

[MacOS] - Installer *Homebrew*, un gestionnaire de paquets, qui permet de faire l'équivalent de *apt* sous Linux. Copier coller dans un terminal la ligne de commande suivante :

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Deuxième étape - Installation de *Python3* et de *pip*

1. **[Linux / Windows]** - Mise à jour puis à niveau du système de commande apt (*Python3* est normalement préinstallé) :

```
sudo apt update
sudo apt -y upgrade
sudo apt install python3-pip
```

   [MacOS] - Mise à jour puis à niveau du système de commande brew et installer *Python3* :

```
brew update
brew install python
```

2. **[Linux / Windows / MacOS]** - Installation des packages *Python* utilisés dans le TP

```
pip install numpy
pip install matplotlib
pip install numba
```

Si cela ne marche pas sous **Mac** car vous avez un Python2 déjà installé, faites les mêmes commandes avec `pip3`.

Troisième étape - Installation du compilateur C-C++ *clang* :
[Linux / Windows]

```
sudo apt install clang
```

[MacOS] Suivre les étapes proposées ici Installation de *Clang*, puis de *OpenMP* pour faire du parallélisme et enfin *ImageMagick* pour sauvegarder une image au format JPEG :

```
command xcode-select --install
brew install libomp
brew install imagemagick
```

Quatrième étape - Installation de l'outil de mesure de la consommation du processeur :
[MacOS ou Windows avec processeur Intel] : Il s'agit du logiciel *Intel Power Gadget* à télécharger en suivant ce lien.
[MacOS avec processeur ARM] : Il s'agit du logiciel *MX Power Gadget* à télécharger en suivant ce lien.
[Windows avec processeur ARM] : Il s'agit du logiciel *HWinfo* à télécharger en suivant ce lien.
[Linux] : Il s'agit de *s-tui* à installer via *pip* :

```
sudo pip install s-tui
```