TP1 Premiers pas avec OpenMP

M1 informatique, Université d'Orléans 2021/2022

L'objectif de ce TP est de découvrir OpenMP (https://www.openmp.org/) en expérimentant les différentes directives depuis les plus bas niveaux jusqu'aux constructions les plus sophistiquées autour des boucles.

0. Fiches de TP

Les énoncés des TP sont fournis sous forme de *notebook Jupyter*. Il s'agit de documents modifiables. Lorsque le document est visualisé dans un navigateur web, des menus (et éventuellement une barre d'outil si elle est activée) permettent de manipuler le document.

Le document est composé d'une suite de cellules qui peuvent contenir du texte mis en forme au format Markdown (https://medium.com/analytics-vidhya/the-ultimate-markdown-guide-for-jupyter-notebook-d5e5abf728fd), du code Python ou encore du texte brut. Ces cellules peuvent être *déplacées*, de nouvelles cellules **insérées**, etc.

```
Q0. Les énoncés comportent des zones question (comme celle-ci) dans
lesquelles écrire vos réponses. Prenez le temps de répondre à chacune
de ces questions, c'est le meilleur moyen de bien assimiler le TP.

Si ce n'est pas déjà fait, activez l'affichage de la barre d'outils.

En observant les menus et le code source des cellules précédentes,
expliquez comment il est possible :
 - d'insérer une cellule de texte Markdown :

 - d'exécuter une cellule Markdown pour passer du code source à la
vue formattée :

 - de mettre en gras un passage :

 - d'insérer un lien vers une page web :
```

PS. Pour ceux d'entre nous qui préférent travailler en mode texte ou dans un terminal, il est possible de travailler sur le source de la fiche de TP. On perd simplement la possibilité d'exécuter directement du code Python au fil de l'eau dans la fiche.

1. Hello parallel world!

🛟 (https://basthon.fr/doc.html)

Pour cette première partie, il suffit de disposer d'un compilateur C++ avec support pour OpenMP, d'un terminal ouvert et d'un éditeur de fichiers.

Commençons par créer un fichier `Makefile` pour préciser quel compilateur utiliser et quels arguments lui passer à la compilation. Nous n'écrirons pas de règles, notre source n'ayant aucune dépendance, les règles implicites suffiront pour compiler notre programme.

```
CXX=g++
CXXFLAGS=-std=c++11 -Wall -Wextra -pedantic -fopenmp
```

Et voici notre premier programme OpenMP `hello.cpp`

```
~~~c++
#include <iostream>
#include <omp.h>
using namespace std;

int main(int, char *[])
{
    cout << "Bonjour" << endl;
    cout << "Nous sommes " << omp_get_num_threads() << " threads dans
cette équipe" << endl;
    cout << "Au revoir" << endl;
}
~~~
```

Si tout se passe bien, la commande `make hello` devrait compiler ce programme et sans surprise au lancement on obtient un comportement très séquentiel.

```
$ ./hello
Bonjour
Nous sommes 1 threads dans cette équipe
Au revoir
```

```
Q1. Relancez la commande en plaçant `TRUE` dans la variable
d'environnement `OMP_DISPLAY_ENV` et copiez ici le résultat. Ce sont
les valeurs par défaut des paramètres configurables depuis
l'environnement d'exécution.

```
$ OMP_DISPLAY_ENV=TRUE ./hello
```

OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '4'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'FALSE'
```

(https://basthon.fr/doc.html)

```
  OMP_PLACES = ''
  OMP_STACKSIZE = '0'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
  OMP_DISPLAY_AFFINITY = 'FALSE'
  OMP_AFFINITY_FORMAT = 'level %L thread %i affinity %A'
OPENMP DISPLAY ENVIRONMENT END
Bonjour
Nous sommes 1 threads dans cette équipe
Au revoir
```

1.1. pragma omp parallel

La directive `#pragma omp parallel` indique que l'instruction (ou le bloc d'instructions) qui suit doit être exécuté par une équipe de *threads* (souvenez-vous qu'openMP repose sur le paradigme de programmation parallèle fork-join (https://en.wikipedia.org/wiki/Fork%E2%80%93join_model)).

```
Q2. Ajoutez la directive `#pragma omp parallel` juste avant la ligne
qui affiche `Bonjour` puis recompilez et exécutez le programme
`hello`. Quelle sortie obtenez-vous ? Pourquoi autant de fois
`Bonjour` et si peu de fois `Au revoir` ?

BonjourBonjour
Bonjour

Bonjour
Nous sommes 1 threads dans cette équipe
Au revoir

Tous les threads executent la premiere instruction et le reste par ur
seul



Modifiez le programme pour que ce soit tout le bloc d'affichage
`Bonjour` + `Au revoir` qui soit parallélisé. Quel affichage obtenez-
vous ? Reste-t-il identique si vous relancez le programme ?

BonjourBonjour
Nous sommes 4 threads dans cette équipe
Au revoir
Bonjour
Nous sommes 4 threads dans cette équipe
Au revoir
Bonjour
Nous sommes 4 threads dans cette équipe
Au revoir

Nous sommes 4 threads dans cette équipe
```

```
Au revoir

Ils accèdent tous a la sortie de manière désordonnée
```

La fonction de bibliothèque `omp_get_num_threads` permet de connaître le nombre total de *threads* dans l'équipe et la fonction `omp_get_thread_num` indique le numéro du *thread* courant.

La variable d'environnement `OMP_NUM_THREADS` permet de spécifier le nombre de *threads* souhaités. *Il est aussi possible d'utiliser la fonction de la bibliothèque `omp_set_num_threads` mais l'utilisation d'une variable d'environnement permet de changer la valeur plus facilement sans recompiler le programme.*

Les variables déclarées à l'intérieur d'un bloc parallèle sont par défaut privées : chaque thread dispose de sa propre variable. Les variables déclarées à l'extérieur d'un bloc parallèle et celles allouées sur le tas sont partagées par tous les threads de l'équipe.

```
Q3. Modifiez votre programme pour que chaque thread stocke son
identifiant et le nombre total de *threads* dans des variables `id`
et `np` puis affichez `Bonjour(13/54)` puis `Au revoir(13/54)` où
`54` est remplacé par le nombre de *threads* de l'équipe et `13` par
le numéro du *thread* courant. Quelle commande pour lancer le
programme avec 32 *threads* ? Quel affichage obtenez-vous ?
```

In [1]:
```
~~~c++
pragma omp parallel
    {
        int nb = omp_get_num_threads();
        cout << "Bonjour (" << omp_get_thread_num() << "/"<<nb<<")"<<
        cout << "Au revoir(" << omp_get_thread_num() << "/"<<nb<<")"<
    }
~~~
```

```
  File "<input>", line 1
    ~~~c++
        ^
SyntaxError: invalid syntax
```

In [2]:
```
OMP_NUM_THREADS=32 ./hello
=> 32 threads
```

```
  File "<input>", line 1
    OMP_NUM_THREADS=32 ./hello
                       ^
SyntaxError: invalid syntax
```

In [ ]:
```
Bonjour (Bonjour (0/4)
Au revoir(Bonjour (0/4)
3/4)
Au revoir(3/4)
Bonjour (1/4)
Au revoir(1/4)
2/4)
Au revoir(2/4)
```

(https://basthon.fr/doc.html)

1.2. pragma omp critical, single, barrier

La directive `#pragma omp critical` définit une section critique, une portion de code qui ne peut être accédée par les *threads* qu'un seul à la fois.

```
Q4. Modifiez votre programme pour que les affichages ne s'entremêlent
pas : associer une section critique à chaque ligne d'affichage.
```

```
~~~
int nb = omp_get_num_threads();
        #pragma omp critical
        cout << "Bonjour (" << omp_get_thread_num() << "/"<<nb<<")"<<
endl;
        #pragma omp critical
        cout << "Au revoir(" << omp_get_thread_num() <<
"/"<<nb<<")"<< endl;
~~~
```

```
Bonjour (1/4)
Au revoir(1/4)
Bonjour (2/4)
Au revoir(2/4)
Bonjour (0/4)
Au revoir(0/4)
Bonjour (3/4)
Au revoir(3/4)
```

In [ ]:

La directive `#pragma omp barrier` définit une barrière de synchronisation, un rendez-vous que tous les *threads* doivent avoir rejoint avant d'être autorisés à poursuivre leur exécution.

```
Q5. Modifiez votre programme pour que personne ne dise `au revoir`
avant que tout le monde n'ait terminé de dire `bonjour` !
```

'''int nb = omp_get_num_threads();

```
        #pragma omp critical
        cout << "Bonjour (" << omp_get_thread_num() << "/"<<nb
    <<")"<< endl;
        //cout << "Nous sommes " << omp_get_num_threads() << " t
    hreads dans cette équipe" << endl;
        //cout << "Info donnée par le thread numéro : " << omp_g
    et_thread_num() << "/"<<nb<< endl;
        #pragma omp barrier
        cout << "Au revoir(" << omp_get_thread_num() << "/"<<nb
    <<")"<< endl;
        '''
```

(https://basthon.fr/doc.html)

In [ ]:
```
Bonjour (0/4)
Bonjour (2/4)
Bonjour (1/4)
Bonjour (3/4)
Au revoir(2Au revoir(3/4)/4)
Au revoir(1/4)

Au revoir(0/4)
```

La directive `#pragma omp single` définit une section de code qui sera exécutée par un unique *thread*

Q6. Modifiez votre programme pour qu'un seul thread annonce le nombre de threads de l'équipe une fois que tout le monde a dit `bonjour` et qu'un second thread (pas nécessairement le même) annonce la fin du TP ensuite, avant que tout le monde dise `au revoir`. N'oubliez pas de faire en sorte que ces threads qui parlent annoncent leur `id`.

```
int main(int, char *[])
{

    #pragma omp parallel
    {
        int nb = omp_get_num_threads();
        #pragma omp critical
        cout << "Bonjour (" << omp_get_thread_num() << "/"<<nb<<")"<<
endl;
        #pragma omp barrier
        #pragma omp single
        cout<< "Nous sommes : "<<nb<< endl;
        #pragma omp single
        cout<< "C'est bientot la fin du TP "<< endl;
        #pragma omp barrier
        #pragma omp critical
        cout << "Au revoir(" << omp_get_thread_num() <<
"/"<<nb<<")"<< endl;
    }
}
```

In [ ]:
```
Bonjour (3/4)
Bonjour (2/4)
Bonjour (1/4)
Bonjour (0/4)
Nous sommes : 4
C'est bientot la fin du TP
Au revoir(2/4)
Au revoir(0/4)
Au revoir(1/4)
Au revoir(3/4)
```

1.3. On récapitule et on mesure le temps de calcul !

Voici un programme `syracuse.cpp` que nous allons paralléliser. Il utilise la fonction de bibliothèque `omp_get_wtime` pour mesurer le temps réel écoulé entre deux points du programme.

(https://basthon.fr/doc.html)

```
#include <iostream>
#include <omp.h>
using namespace std;

#define TOTAL 2000000

int main(int, char *[])
{
    int pass=0, cur;
    double start = omp_get_wtime();
////////////////////// MODIFIER A PARTIR D'ICI UNIQUEMENT
//////////////////////
    for(int i=0; i<TOTAL; i++) {
        cur=i;
        while (cur>1)
            cur=cur%2?3*cur+1:cur/2;
        pass++;
    }
////////////////////// MODIFIER JUSQU'ICI UNIQUEMENT
//////////////////////
    double end = omp_get_wtime();
    cout << pass << " out of " << TOTAL << "! (delta=" << TO
TAL-pass << ")" << endl;
    cout << "ellapsed time: " << (end-start)*1000 << "ms" <<
endl;
}
```

Q7. Compilez et exécutez le programme. Vérifiez que la variable `pass` doit être égale à `TOTAL` en fin de programme. Si besoin, modifier la valeur de `TOTAL` pour avoir un temps d'exécution de l'ordre de `1000ms`.

Q8. En utilisant **uniquement les directives vues ci-dessus**, parallélisez ce programme. Attention, le comportement du programme doit toujours être le même, il s'agit de répartir le travail entre plusieurs *threads* ! Si besoin, utilisez des sections critiques pour protéger l'accès aux variables partagées. Attention à la variable `cur` qui mérite sans doute une version privée par `thread` !

Si votre programme effectue une section critique à chaque passage dans la boucle... il est certainement peu efficace et la parallélisation peut même aller jusqu'à dégrader le temps de calcul par rapport au séquentiel !

Q9. Ajoutez une variable privée `mycur` par thread et effectuez une unique section critique pour mettre à jour `pass` après la boucle. Exécutez le programme et observez le gain de temps ! L'accélération obtenue est-elle égale au nombre de threads ? Pourquoi ?

(https://basthon.fr/doc.html)

In [ ]:
```
#pragma omp parallel
    {
        int passres=0; //pass local que l'on ajoute à la fin des iter
        int curlocal; //curr local pour éviter colisions
        int id = omp_get_thread_num();
        int num = omp_get_num_threads();
        for (int i = id; i < TOTAL; i+=num) {
            curlocal = i;
            while (curlocal > 1)
                curlocal = curlocal % 2 ? 3 * curlocal+ 1 : curlocal

            passres++;
        }
        #pragma omp critical
        pass+=passres;

    }
```

PS. On pourra s'aider d'une boucle shell comme ceci :

```
$ for np in 1 2 4 8 16; do OMP_NUM_THREADS=$np ./syracuse; d
one
```

In [ ]:
```
2000000 out of 2000000! (delta=0)
ellapsed time: 1238.74ms
2000000 out of 2000000! (delta=0)
ellapsed time: 727.381ms
2000000 out of 2000000! (delta=0)
ellapsed time: 455.294ms
2000000 out of 2000000! (delta=0)
ellapsed time: 442.659ms
2000000 out of 2000000! (delta=0)
ellapsed time: 426.407ms
```

```
CHECK-POINT #1 <<<<<<< LORSQUE VOUS ATTEIGNEZ CETTE ENDROIT DE LA
FICHE, APPELEZ VOTRE CHARGÉ DE TP ! >>>>>>>>
```

2. Vers l'infini et au-delà

Dans cette deuxième partie, nous allons paralléliser un code existant. Téléchargez et décompressez l'archive `astro.tar.gz` fournie avec le TP. **Lire le fichier `README.md`** — dans un premier temps, vous pouvez laisser de côté la partie `bench` et `show` qui servira à effectuer des mesures de performance.

Le programme à paralléliser prend en entrée une image au format FITS (https://fr.wikipedia.org/wiki/Flexible_Image_Transport_System) Astro et produit en sortie une image Resca au format PGM (https://fr.wikipedia.org/wiki/Portable_pixmap) produite par une mise à l'échelle comme ceci :

🛟 (https://basthon.fr/doc.html)

$$\forall i, j, \quad \text{Resca}(i,j) = 255\frac{\text{Astro}(i,j) - m}{M - m} \qquad \text{où} \begin{cases} M = \max_{i,j} \text{Astro}(i,j) \\ m = \min_{i,j} \text{Astro}(i,j) \end{cases}$$

Ces calculs sont effectués dans la partie du code qui est à modifier :

```
    ////////////////////// MODIFIER A PARTIR D'ICI UNIQUEMENT
////////////////
    for (j = 0; j < astro.height(); j++)
      for (i = 0; i < astro.width(); i++) {
        amin = min(amin, astro(i, j));
        amax = max(amax, astro(i, j));
      }
    unsigned short arange = amax - amin;
    for (j = 0; j < astro.height(); j++)
      for (i = 0; i < astro.width(); i++)
        resca(i, j) = (astro(i, j) - amin) * 255 / arange;
    ////////////////////// MODIFIER JUSQU'ICI UNIQUEMENT
////////////////////
```

Q10. Compilez et exécutez le programme `linear` en suivant les instructions du fichier `README.md`. Visualisez l'image obtenue et copiez là dans un fichier `resca-ref.pgm`.

Dans le code ci-dessus il y a deux boucles. Laquelle vous semble la plus facile à paralléliser ? Quel est le rôle des variables `amin`, `amax` et `arange` ?

Q11. En utilisant **uniquement les directives vues en partie 1**, parallélisez ce programme en créant deux régions parallèles, une autour de chaque bloc de boucles. Testez votre programme avec `10` itérations et comparez son temps de calcul à la version séquentielle.

Vérifiez que l'image produite est bien identique à la version séquentielle !

Astuce. Pour comparer les deux images `resca-ref.pgm` et `resca.pgm`, vous pouvez utiliser par exemple l'outil de comparaison de *Image Magick* ou *Graphics Magick* si vous en disposez :

```
$ compare /tmp/resca-ref.pgm /tmp/resca.pgm /tmp/diff.png &&
xdg-open /tmp/diff.png
$ magick compare /tmp/resca-ref.pgm /tmp/resca.pgm /tmp/dif
f.png && xdg-open /tmp/diff.png
$ gm compare /tmp/resca-ref.pgm /tmp/resca.pgm /tmp/diff.png
&& xdg-open /tmp/diff.png
```

Sinon vous pouvez de manière plus minimaliste et comparer les empreintes des deux

(https://basthon.fr/doc.html)

fichiers pour tester leur égalité :

```
$ shasum /tmp/resca-ref.pgm /tmp/resca.pgm
```

2.1. pragma omp for, collapse

Les boucles sont tellement fréquentes dans le code numérique à paralléliser qu'OpenMP dispose de directives dédiées à cette tâche. Ainsi `#pragma omp for` distribue les indices de la boucle qui suit la directive entre les différents *threads* de l'équipe et nous libère des fastidieux calculs associés.

```
Q12. Simplifiez votre code parallèle pour utiliser un `#pragma omp
for` devant les boucles indicées par `j` de chaque région. OpenMP
s'occupe de privatiser la variable de boucle mais attention aux
autres variables ! *En particulier il faut veiller à calculer le
minimum et le maximum correctement avec une section critique.*

Vérifiez que l'image produite est bien identique à la version
séquentielle !
```

Lorsque plusieurs boucles sont imbriquées, la clause `collapse(N)` permet de paralléliser simultanément les `N` premiers niveaux de boucles. Dans notre exemple il peut être pertinent de traiter les deux boucles simultanément.

```
Q13. Modifiez votre programme pour paralléliser les deux niveaux de
boucle simultanément.

Vérifiez que l'image produite est bien identique à la version
séquentielle !
```

2.2. reduction

Notre premier bloc de boucles imbriquées calcule un minimum et un maximum. Effectuer ce calcul dans chaque *thread* avant de mettre en commun le résultat est une opération très courante. OpenMP permet d'automatiser cette phase grâce à la clause `reduction(op:var)` où `op` est l'opération qu'on souhaite réduire et `var` la variable où stocker le résultat.

```
Q14. Modifiez votre programme pour simplifier le calcul de `amin` et
`amax` grâce à `reduction`.

Vérifiez que l'image produite est bien identique à la version
séquentielle !
```

Enfin, il est possible de contracter les directives `#pragma omp parallel` et `#pragma omp for` lorsqu'elles se suivent en un unique `#pragma omp parallel for` .

(https://basthon.fr/doc.html)

Q15. Modifiez votre programme pour que la seule différence entre la version séquentielle et la version parallèle soit l'ajout d'une directive en tête de chaque bloc de boucles.

Vérifiez que l'image produite est bien identique à la version séquentielle !

2.3. schedule

La manière dont les *threads* se partagent les indices impacte les performances du calcul. OpenMP permet de choisir une stratégie de distribution des indices à travers la clause `schedule(S)` où `S` décrit la politique à appliquer. Les stratégies les plus classiques sont :

- `schedule(static)` : découpe les indices en autant de blocs de même taille qu'il y a de *threads* ;
- `schedule(static,N)` : découpe les indices en blocs de taille `N` et les distribue équitablement et cycliquement entre les *threads* ;
- `schedule(dynamic,N)` : découpe les indices en blocs de taille `N` et les attribue successivement aux *threads* au fur et à mesure qu'ils sont disponibles ;
- `schedule(runtime)` : applique la stratégie décrite dans la variable d'environnement `OMP_SCHEDULE` .

À l'aide des deux variables `OMP_SCHEDULE` et `OMP_NUM_THREADS` , il est possible d'étudier l'influence de la stratégie de distributions d'indices et de chercher les meilleurs paramètres pour une machine donnée. C'est ce que proposent les scripts `bench` , pour collecter des statistiques, et `show` , pour afficher ces données sous forme de courbes de performance.

Si T_S est le temps de calcul séquentiel et $T_P(p)$ le temps de calcul parallèle pour effectuer une tâche avec p *threads*, on définit l'accélération comme la quantité $S^*(p) = \frac{T_S}{T_P(p)}$ et l'efficacité comme $E^*(p) = \frac{S^*(p)}{p}$.

Q16. Modifiez votre programme pour utiliser `schedule(runtime)` sur les deux régions parallèles. Lire la fin du fichier `README.md`. Utilisez les scripts `bench` et `show` pour produire des courbes pertinentes. Il conviendra d'adapter le nombre de *threads*, le nombre d'itérations et l'image utilisée à votre machine.

Si vous ne disposez pas des bibliothèques pour générer les courbes, vous pouvez les générer dans ce *notebook* en collant le contenu de `stats.csv` puis en exécutant les cellules ci-dessous.

In [ ]:
```
from io import StringIO
from ezplot import ezplot
```

In [ ]:
```
data="""
### COLLER A LA PLACE DE CETTE LIGNE LE CONTENU DE stats.csv
"""
```

In [ ]:
```
ezplot(StringIO(data),['bs'])
```

(https://basthon.fr/doc.html)

In [ ]:
```
ezplot(StringIO(data), ['schedule'])
```

```
CHECK-POINT #2 <<<<<<< LORSQUE VOUS ATTEIGNEZ CETTE ENDROIT DE LA
FICHE, APPELEZ VOTRE CHARGÉ DE TP ! >>>>>>>>
```

Références

1. Spécification OpenMP (https://www.openmp.org/specifications/)
2. OpenMP Reference Guide (https://www.openmp.org/resources/refguides/)
3. Exemples de statistiques collectées pour la partie 2 (?from=progpar/perf.ipynb&module=progpar/ezplot.py) !