Programmation Parallèle

Fiche 1 : Introduction à OpenMP

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

1 Création d'une équipe de threads

La directive `#pragma omp parallel` indique que l'instruction ou le bloc d'instructions suivant cette directive doit être réalisé par une *équipe* de threads. On appelle *section parallèle* l'instruction ou le bloc concerné par la directive.

```
1 int main()
2 {
3
4   printf("Bonjour !\n");
5   printf("Au revoir !\n");
6
7   return 0;
8 }
```

1. Modifiez le programme `hello.c` en ajoutant une directive `#pragma omp parallel` en ligne 3.
2. Compilez puis exécutez le programme de la façon suivante :

   ```
   gcc -o hello hello.c
   ./hello
   ```

3. Recompilez en donnant cette fois l'option `-fopenmp` à `gcc`. Exécutez le programme plusieurs fois. Combien de threads sont utilisés ? Comparez ce nombre au nombre de coeurs (logiques) de votre machine. À quel moment s'affiche le message `Au revoir !` ?
4. La variable d'environnement `OMP_NUM_THREADS` permet de contrôler le nombre de threads utilisés par défaut. Lancez l'exécution en entrant la commande :

   ```
   OMP_NUM_THREADS=13 ./hello
   ```

5. La fonction `omp_get_thread_num` retourne le numéro d'équipier (l'identité) du thread qui l'appelle. Sa mise en œuvre nécessite l'inclusion du fichier d'entête OpenMP : `#include <omp.h>`. Modifiez les deux appels à `printf` de sorte à afficher également le numéro du thread appelant. Compilez puis exécutez plusieurs fois le programme en faisant varier le nombre de threads. L'ordre d'exécution est-il toujours le même ?
6. Faites en sorte que chaque thread affiche également le message `Au revoir !` en englobant dans une seule section parallèle les deux appels à `printf`. Testez le programme avec une bonne douzaine de threads.

7. La directive `#pragma omp barrier` permet à une équipe de threads de se fixer un rendez-vous (de s'attendre) à une ligne de code donnée. Insérer cette directive entre les deux appels à `printf`. Compilez et testez le programme.

2 Section critique

La directive `#pragma omp critical` impose que chaque thread exécute l'instruction ou le bloc d'instructions suivant en solitaire. Le code gardé par la directive `critical` est appelé *section critique* et on dit que la section critique est exécutée en *exclusion mutuelle*. Utiliser cette directive pour faire en sorte que les affichages des différents threads ne s'entremêlent pas (note : on supprimera la barrière placée lors de l'exercice précédent).

3 Variable partagée vs variable privée

Dans le programme `partage.c` on déclare `k` et `i` deux variables locales à la fonction `main`.

```
1  int main()
2  {
3    int k = 0;
4
5  #pragma omp parallel
6    {
7      int i;
8      for(i = 0; i < 100000; i++)
9        k++;
10   }
11
12   printf("nbthreads x 100000 = %d\n",k);
13   return 0;
14 }
```

La variable `k` est déclarée avant l'appel à la directive `parallel`, cette variable sera partagée entre les différents threads : tout thread de la section parallèle suivante pourra consulter / modifier à loisir cette variable.

La variable `i` est déclarée à l'intérieur de la section parallèle, c'est une variable privée du thread : chaque thread aura son propre exemplaire et ne pourra pas modifier les autres.

1. Compilez puis exécutez le programme `partage.c`. On observe que la valeur de `k` (affichée à la fin du programme) est exceptionnellement égale à 100 000 fois le nombre de threads utilisés. Les incrémentations `k++` ne semblent pas toutes avoir été exécutées et la valeur finale de `k` change d'une exécution à l'autre. *Manipuler sans précaution une variable partagée conduit à un résultat indéterminé.*
2. Insérez une section critique en ligne 9 pour ne plus perdre d'incrémentation, compilez, testez.

4 Distribution d'une boucle for

On cherche à rendre parallèle l'exécution du code suivant où l'on sait que les différents appels à `traitement(i)` sont indépendants les uns des autres :

```
1 for(int i = 0; i < N; i++)
2    traitement(i);
```

Pour cela il s'agit de distribuer (équitablement) les indices de la boucle **`for`** aux différents threads en faisant en sorte que chaque indice soit traité exactement une fois. Voici comment réaliser automatiquement cette distribution en OpenMP :

```
1 #pragma omp parallel
2 #pragma omp for schedule(static)
3   for(int i = 0; i < N; i++)
4      traitement(i);
```

4.1 Politiques de distribution des indices d'une boucle

La directive *`#pragma omp for`* indique que les indices de la boucle qui suit sont à répartir entre les threads et la clause `schedule` précise l'algorithme de distribution à utiliser. Il y a quatre principaux types de distribution :

- `schedule(static)` indique que les indices sont à distribués équitablement en autant de blocs d'indices consécutifs qu'il y a de threads ;
- `schedule(static, k)` indique que les indices doivent être distribués cycliquement par blocs de k indices.
- `schedule(dynamic, k)` indique que les indices doivent être distribués à la demande par blocs de k indices.
- `schedule(guided, k)` pour les curieux.

Notons enfin que la directive *`#pragma omp parallel for`* est la contraction de l'enchaînement des directives *`#pragma omp parallel`* et *`#pragma omp for`*.

Étudions ces différentes politiques à l'aide du programme `boucle-for.c` :

```
1 int main()
2 {
3   int i;
4
5   for(i=0; i < 40; i++)
6     printf("%d traite %i\n",omp_get_thread_num(),i);
7
8   return 0;
9 }
```

1. Tout d'abord placez une directive *`#pragma omp parallel for`* en ligne 4. Compilez et testez plusieurs fois. On pourra limiter à 4 le nombre de threads utilisés.

2. Modifiez le programme pour tester différentes politiques de distribution. Utilisez la commande :

```
OMP_NUM_THREADS=4 ./boucle-for | sort -n -k 3
```

pour visualiser plus facilement le travail réalisé par chaque thread.

4.2 Calcul de la somme d'un tableau - réduction

Nous avons vu qu'il était nécessaire de protéger les accès aux variables partagées. Ceci peut se faire à l'aide de sections critiques mais aussi à l'aide d'instructions atomiques. Une section atomique est une section critique réduite à une instruction. Nous allons comparer le coût de différentes techniques de protection sur le code suivant :

```
1 for (i=0; i < N; i++)
2   sum += tab[i];
```

Remplacez les `TODO` du programme `sum.c` par les techniques suivantes :

1. section *`#pragma omp parallel for`* où la variable `sum` partagée est accédée en section critique ;
2. section *`#pragma omp parallel for`* où la variable `sum` partagée est accédée en section atomique *`#pragma omp atomic`* ;
3. utilisation d'une variable locale `ma_somme` dans laquelle chaque thread accumule des valeurs de son côté durant la boucle, puis ajoute sa contribution à `sum` en une fois ;
4. réduction *`#pragma omp parallel for reduction(+:sum)`* : le compilateur transforme le programme pour introduire une variable locale afin de minimiser le temps passé en section critique.

5 EasyPAP

Nous allons maintenant nous intéresser à des traitements qui manipulent des images, qui sont des tableaux en 2 dimensions qui contiennent des pixels.

Pour faciliter le développement, nous allons utiliser un environnement nommé EASYPAP permettant d'afficher des images, de lancer des calculs dessus, et de visualiser interactivement les évolutions de l'image à chaque itération. Suivez les indications pour récupérer EASYPAP sur cette page :

https://gforgeron.gitlab.io/pap/td/

Une fois l'archive récupérée et décompressée, placez vous dans le répertoire `easypap` et suivez les instructions des premières pages de la documentation (pour compiler l'environnement et lancer vos premières exécutions). La documentation se trouve ici :

https://gforgeron.gitlab.io/easypap/

La lecture des premières pages vous sera très utile !

Si tout va bien, vous devriez pouvoir observer une image animée lorsque vous tapez :

```
./run -k spin
```

5.1 Découverte du code

Le fichier `kernel/c/spin.c` contient plusieurs variantes du noyau `spin`. Commençons par modifier (temporairement) la variante séquentielle pour bien comprendre la façon dont l'image est parcourue.

Appliquez la modification suivante (Figure 1, ligne 7) dans le code de la fonction `spin_compute_seq`.

```
1  unsigned spin_compute_seq (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5      for (int i = 0; i < DIM; i++)
6        for (int j = 0; j < DIM; j++)
7          if (j < DIM/2)  // Try also: if (i < DIM/2 || j < DIM/2)
8          cur_img (i, j) = compute_color (i, j);
9
10     rotate (); // Slightly increase the base angle
11   }
12
13   return 0;
14 }
```

FIGURE 1 – Version séquentielle simple du noyau `spin` (`kernel/c/spin.c`).

Recompilez et relancez pour vérifier l'effet de votre modification :

```
./run --kernel spin --variant seq
```

Regardons à présent la variante `tiled` (Figure 2), qui découpe virtuellement l'image en tuiles rectangulaires de taille `TILE_W` × `TILE_H` pixels.

```
1  unsigned spin_compute_tiled (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5      for (int y = 0; y < DIM; y += TILE_H)
6        for (int x = 0; x < DIM; x += TILE_W)
7          if ((x / TILE_W + y / TILE_H) % 2)
8          do_tile (x, y, TILE_W, TILE_H);
9
10     rotate ();
11   }
12
13   return 0;
14 }
```

FIGURE 2 – Version tuilée du noyau `spin`.

L'appel à `do_tile (int x, int y, int w, int h)` se charge de calculer les pixels à

l'intérieur du rectangle défini par les paramètres. Cette fonction est en fait un aiguilleur vers la fonction `spin_do_tile_default` par défaut.

Appliquez la modification suggérée (Figure 2, ligne 7) dans le code de la fonction `spin_compute_tiled`, recompilez et vérifiez l'effet à l'écran :

```
./run --kernel spin --variant tiled
```

Faites variez la taille des tuiles en utilisant l'option `--tile-size` suivie d'une puissance de 2. Par exemple :

```
./run --kernel spin --variant tiled --tile-size 8
```

Pour définir des tuiles rectangulaire, utilisez les options `--tile-width` et `--tile-height` :

```
./run --kernel spin --variant tiled --tile-width 64 --tile-height 8
```

Vous pouvez utiliser l'option `--monitoring` (ou `-m` en version courte) pour afficher des fenêtre de *monitoring* qui vous permettrons respectivement de voir l'activité des threads du programme ainsi que les zones de l'image sur lesquelles ils travaillent :

```
./run --kernel spin --variant tiled --tile-width 64 --tile-height 8 --monitoring
```

Prenez un moment pour contempler... puis retirez la ligne 7.

5.2 Première version OpenMP

Il s'agit maintenant d'écrire des versions parallèles du noyau `spin` en utilisant OpenMP.

Pour cela, recopiez la fonction `spin_compute_tiled` vers une nouvelle variante que vous appellerez `spin_compute_omp`.

```
1  unsigned spin_compute_omp (unsigned nb_iter)
2  {
3    for (unsigned it = 1; it <= nb_iter; it++) {
4
5  #pragma omp parallel for
6      for (int y = 0; y < DIM; y += TILE_H)
7        for (int x = 0; x < DIM; x += TILE_W)
8          do_tile (x, y, TILE_W, TILE_H);
9
10     rotate ();
11   }
12
13   return 0;
14 }
```

FIGURE 3 – Version OpenMP simple du noyau `spin`.

Effectuez les modifications suggérées sur la figure 3 et recompilez. Normalement, vous devez maintenant pouvoir essayer votre version parallèle :

```
./run -k spin -v omp -m
```

Que constatez-vous ? Essayez différentes tailles de tuiles. Vous pouvez également modifier le nombre de *threads* utilisés par OpenMP au moyen de la variable `OMP_NUM_THREADS`. Par exemple :

```
OMP_NUM_THREADS=12 ./run -k spin -v omp -m -ts 4
```

5.3 Visualisation des différentes stratégies d'attribution des indices de boucles

Vous êtes désormais en mesure d'expérimenter différentes stratégies d'ordonnancement des indices de boucles en observant la répartition des pixels par CPU sur la fenêtre de *monitoring* des tuiles.

Par exemple, vous pouvez ajouter en ligne 5 la clause `schedule (static,1)`, recompiler, et visualiser le changement en relançant :

```
./run -k spin -v omp -m -ts 4
```

Pour jouer avec toutes les stratégies d'ordonnancement sans nécessairement recompiler le code entre chaque exécution, vous pouvez plutôt utiliser la clause `schedule (runtime)` en ligne 5, ce qui vous permet de specifier la stratégie à utiliser au moyen de la variable d'environnement `OMP_SCHEDULE`. Ainsi, une fois le code recompilé, vous pouvez exécuter successivement :

```
OMP_SCHEDULE=static ./run -k spin -v omp -m -ts 1

OMP_SCHEDULE=static,1 ./run -k spin -v omp -m -ts 1

OMP_SCHEDULE=dynamic ./run -k spin -v omp -m -ts 1
```

5.4 Mesure de l'accélération

Pour déterminer l'accélération obtenue, c'est-à-dire le gain obtenu en passant d'une version séquentielle à une version multithread, on mesure d'abord précisément le temps obtenu par chacune des versions en désactivant l'affichage (option `--no-display` ou `-n`). Par exemple :

```
[my-machine] ./run -k spin -v seq -i 200 -n
Using kernel [spin], variant [seq]
Computation completed after 200 iterations
6203.184

[my-machine] ./run -k spin -v omp -i 200 -n
Using kernel [spin], variant [omp]
Computation completed after 200 iterations
841.775
```

Sur cet exemple, l'accélération est de $\frac{6203}{841} \approx 7.375$. Calculez celle obtenue au CREMI.

5.5 Expérimentation

Assurez-vous d'utiliser un environnement python récent (au moins 3.8).

Nous allons utiliser le script `plots/run-xp-spin.py` pour chercher la « meilleure combinaison » (taille de tuile, stratégie de répartition), celle qui a l'accélération la plus importante. Consulter puis lancer ce script python. Un fichier `spin.csv` est produit, vous pouvez le consulter rapidement, puis visualiser son contenu au moyen des commandes :

```
plots/run-xp-spin.py             # lance les expériences
plots/easyplot.py -if spin.csv   # produit un fichier spin.pdf
evince spin.pdf                  # affiche le fichier
```

On obtiendra un meilleur rendu en séparant les courbes selon le type de distribution (représenté par le mot clé `schedule`) utilisé grâce à la la commande suivante :

```
plots/easyplot.py -if spin.csv -v omp -- col=schedule
```

On pourra, lorsqu'on aura un moment de spleen, modifier ce script pour chercher la combinaison de paramètres véritablement optimale.

5.6 Parallélisation en colonne

Déplacez votre `#pragma omp parallel for` juste devant la boucle `for (int x)` et recompilez. Observez que la répartition du travail est désormais effectuée par colonnes :

```
OMP_SCHEDULE=static ./run -k spin -v omp -m -ts 1
```

Pour savoir si cette version est plus efficace que la parallélisation en lignes, effectuez une mesure de temps comparable (en nombre d'itérations) avec les expériences précédentes.

Pour que la comparaison soit *fair-play*, nous allons remettre un `#pragma omp parallel` devant la boucle `for (int y)` et utiliser uniquement `#pragma omp for` devant la boucle `for (int x)`. Testez par exemple :

```
OMP_SCHEDULE=static ./run -k spin -v omp -i 200 -n -ts 1
```

Les résultats varient-ils si on utilise une distribution cyclique ? Testez :

```
OMP_SCHEDULE=static,1 ./run -k spin -v omp -i 200 -n -ts 1
```

5.7 Collapse

Dupliquez la fonction `spin_compute_omp` pour créer une fonction `spin_compute_omp_tiled`. Modifiez le code en ajoutant la clause `collapse(2)` à la directive OpenMP `for`. De façon générale la clause `collapse(n)` permet de distribuer des n-uplets d'indices plutôt que de simples indices en « fusionnant » n boucles imbriquées.

Observez l'influence de cette clause sur la distribution des tuiles en faisant varier la politique de distribution.

5.8 Calcul non-homogène sur l'image

Le noyau `mandel` affiche une représentation graphique de l'ensemble de Mandelbrot (https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot) qui est une fractale définie comme l'ensemble des points du plan complexe pour lesquels les termes d'une suite ont un module borné par 2.

À chaque itération, le programme affiche une image dont les pixels ont une couleur qui dépend de la convergence de la suite associée au point complexe du plan. Entre chaque itération, un (dé)zoom est appliqué afin de changer légèrement de point de vue.

5.8.1 Découverte

Vous pouvez lancer le programme de la façon suivante :

```
./run -k mandel
```

Sans surprise, la fonction `mandel_compute_seq` se trouve dans le fichier `kernel/c/mandel.c`. Pour bien comprendre en quoi ce noyau diffère du noyau `spin`, lancez l'exécution tuilée et appuyez sur la touche « h » durant l'exécution :

```
./run -k mandel -v tiled -ts 16 -m
```

L'appui sur « h » bascule la fenêtre des tuiles en mode *heatmap*, mode dans lequel la luminosité des tuiles est proportionnelle à la durée du calcul. Voir Section 3.1 (page 15) de la documentation. Que constatez-vous ?

Vous pouvez également activer le mode *debug* et observer le nombre d'itérations effectuées pour chaque pixel en bougeant la souris sur l'image :

```
./run -k mandel -v tiled -ts 16 -d d
```

5.8.2 Version parallèle

Inspirez-vous de votre implémentation OpenMP du noyau `spin` pour réaliser la version parallèle du noyau `mandel`, en utilisant la directive `collapse(2)` ainsi que `schedule(runtime)`. On appellera `omp_tiled` la version OpenMP basée sur l'utilisation de tuiles.

Trouvez les paramètres permettant d'avoir un affichage graphique identique à celui de la figure 7 (page 16) de la documentation.

Cela corrobore-t-il votre conclusion précédente ?

5.9 Des traces pour en avoir le coeur net

Nous allons maintenant générer une trace d'exécution sur quelques itérations afin de regarder précisément la durée d'exécution de chaque tuile.

Voici comment procéder :

```
# génération d'imagettes
./run --kernel mandel --variant omp_tiled --thumbnails --iterations 10 --quit
# génération de la trace
./run --kernel mandel --variant omp_tiled --trace --iterations 10 --no-display
# lancement de l'outil de visualisation
./view
```

Jeter un oeil à la table 2 (page 21) de la documentation pour voir comment naviguer dans la trace.

5.9.1 À la recherche de l'optimal

Explorez différentes stratégies de parallélisation pour obtenir une accélération satisfaisante, d'abord en vérifiant interactivement l'activité des CPU, puis en utilisant les commandes suivantes pour générer des courbes d'accélération pour le cas 512×512 :

```
plots/run-xp-mandel.py
plots/easyplot.py -if mandel.csv -v omp_tiled -- col=schedule
evince mandel.pdf
```

Modifiez le script `plots/run-xp-heat-mandel.py` afin d'employer le nombre de threads optimal. Utilisez les commandes suivantes pour générer une *carte de chaleur* permettant d'observer le comportement du programme selon la géométrie des tuiles :

```
plots/run-xp-heat-mandel.py
plots/easyplot.py -if heat-mandel.csv --plottype heatmap \
                  -heatx tilew -heaty tileh -v omp_tiled -- col=schedule
evince heat-mandel.pdf
```

Conclure.