Accélération de calculs 2D avec OpenMP

Résumé

Nous nous intéressons à la parallélisation d'algorithmes itératifs produisant des séquence d'images. Nous traiterons un premier cas où la charge de calcul est constante sur toute l'image. Dans le second cas, certaines zones de l'image demanderont beaucoup plus de calcul que d'autres ; on cherchera alors à uniformiser la charge de travail des différents cœurs.

1 Cadre du mini-projet

Ce mini-projet est à faire en binôme et à rendre pour le lundi 4 février 18h. Votre rapport, sous forme de fichier au format PDF, contiendra les modifications apportées au code, les conditions expérimentales et les graphiques obtenus accompagnés chacun d'un commentaire le décrivant et l'analysant.
Les machines utilisées doivent être des machines du CREMI dotées d'au moins 12 cœurs physiques. Les expériences menées devront être peu bruitées : il est inutile de faire des expériences sur une machine chargée. On pourra comparer différentes configurations matérielles.
Vous placerez ce rapport dans `/net/cremi/pwacreni/PROJET-PAP` le nom de votre rapport sera de la forme : `binome1-binome2-clef-secrete.pdf`
Less sources de l'application sont sous `~rnamyst/etudiants/pmg/mini-projet`

2 Prise en main de l'application

On s'intéresse à des traitements qui manipulent des images existantes, ou qui en créent des nouvelles.
On dispose d'une application qui permet d'afficher des images, de lancer des calculs dessus, et de visualiser interactivement les évolutions de l'image à chaque itération.
Pour compiler l'application :

```
cd fichiers/
make
```

Avant toute chose, nous allons nous doter d'une petite base d'images en établissant un lien symbolique vers un répertoire centralisé :

```
ln -s ~rnamyst/etudiants/pmg/images
```

Ca y est, tout est prêt ! Pour lancer l'application avec une image d'exemple (un appui sur la touche `ESC` permet de quitter l'application) :

```
./2Dcomp -l images/shibuya.png
```

Dans le cas présent, le traitement (que l'on appellera également *noyau*) par défaut s'appelle `none` et consiste à ne rien faire. Par défaut, c'est la variante séquentielle (`seq`) qui est invoquée.
On peut spécifier le noyau à invoyer à l'aide de l'option `-k`, et la variante à l'aide de l'option `-v`. Ainsi, on aurait pu lancer cette première exécution comme ceci :

```
./2Dcomp -l images/shibuya.png -k none -v seq
```

Les fonctions implémentant les différentes variantes d'un noyau se trouvent généralement dans le fichier `src/<noyau>.c`. Regardez le contenu du fichier `src/none.c`, et remarquez que la fonction implémentant la variante séquentielle s'appelle "`none_compute_seq`" (bon OK, elle est vide, mais elle existe).

3 Négatif d'une image

On s'intéresse au noyau `invert` qui calcule le négatif d'une image couleur, c'est-à-dire qui remplace chaque pixel par un pixel résultant du complément binaire de chacune de ses composantes couleur Rouge, Vert et Bleu.
Une version séquentielle du code vous est fournie dans le fichier `src/invert.c`. Regardez comment chaque pixel de l'image est parcouru, à l'aide des deux boucles imbriquées. La variable globale `DIM` indique à la fois la largeur et la hauteur de l'image (qui est toujours carrée). L'accès à un pixel à la i^{eme} ligne et à la j^{eme} colonne s'effectue au travers de l'expression `cur_img(i, j)`.
Si on lance le noyau sans option particulière, l'image va clignoter de manière très rapide puisqu'à chaque itération paire l'image retrouve son état d'origine :

```
./2Dcomp -l images/shibuya.png -k invert
```

Pour avoir le temps d'observer chaque image, il est possible de demander une pause à chaque itération :

```
./2Dcomp -l images/shibuya.png -k invert -p
```

Enfin, pour mesurer précisément les performances du noyau `invert` sans le ralentissement dû à l'affichage, il faut utiliser l'option `-n` (*no display*) conjointement avec l'option `-i <nb>` (nombre d'itérations souhaitées). Par exemple :

```
./2Dcomp -l images/shibuya.png -k invert -n -i 1000
```

Notez le temps d'exécution total affiché.
Il s'agit désormais d'élaborer une version parallèle du noyau `invert`.

1. Écrivez une fonction `invert_compute_omp` implémentant une version parallèle OpenMP dans laquelle plusieurs threads calculeront chacun une bande horizontale différente de l'image.
   Testez que le lancement se déroule bien en vous contentant dans un premier temps d'un `printf` dans la région parallèle. Vérifiez que le programme crée bien le nombre de threads demandés :

   ```
   OMP_NUM_THREADS=8 ./2Dcomp -l images/shibuya.png -k invert -v omp -i 1
   ```

2. Si la sortie standard est rassurante, vous pouvez enlever le `printf` et calculer l'inversion des pixels de l'image. Vérifiez visuellement :

   ```
   ./2Dcomp -l images/shibuya.png -k invert -v omp -i 1
   ```

3. Enfin, comparez les performances avec la version séquentielle :

   ```
   ./2Dcomp -l images/shibuya.png -k invert -v omp -n -i 1000
   ```

   Que pensez vous de l'accélération obtenue ($\frac{temps\ sequentiel}{temps\ parallele}$) ?

Notez que dans ce cas, les threads ne sont créés qu'une seule fois, et enchainent chacun 1000 itérations de manière indépendante. Dans le cas général, il ne sera pas toujours possible d'éviter de resynchroniser les threads au moyen d'une barrière à la fin de chaque itération...

4 Décalage vers le haut

On s'intéresse maintenant à appliquer un décallage d'une ligne vers le haut de toute l'image.
Consultez le contenu du fichier `src/scrollup.c`. Le fichier contient simplement une fonction (`scrollup_compute_seq`) qui code une version séquentielle du noyau de calcul « scrollup ».
Notez que le programme utilise cette fois deux images : on lit les valeurs des pixels (provenant de l'itération précédente) qvec `cur_img(i, j)` et on écrit la nouvelle valeur dans `next_img(i, j)`. Une fois que tous les pixels sont calculés, on inverse le rôle des deux images en appelant `swap_images()`, puis on peut passer à l'itération suivante...
À chaque itération, on lit les pixels de `cur_img` et on modifie ceux de `next_img`. À la fin de l'itération, les rôles sont permutés.
Pour lancer le programme en exécutant ce noyau de calcul, tapez simplement (tapez la touche ESC pour quitter) :

```
./2Dcomp -l images/mini-stars.png -k scrollup -v seq
```

Écrivez une version OpenMP du noyau « scrollup ». Attention, il ne faut appeler `swap_images` qu'une seule fois par itération !
Testez que votre noyau fonctionne bien en forçant l'exécution de plusieurs itérations consécutives entre chaque affichage. Par exemple :

```
./2Dcomp -l images/shibuya.png -k scrollup -v omp -r 4
```

Le programme n'affiche dans ce cas qu'une image sur 4, mais surtout le paramètre `nb_iter` vaut 4 dans ce cas (et non 1 qui est la valeur par défaut en mode interactif).

5 Mandelbrot

Nous allons maintenant nous intéresser à des calculs plus intenses pour chaque pixel, et surtout irréguliers sur la surface de 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 lq suite associée au point complexe du plan. Entre chaque itération, un zoom est appliqué afin de changer légèrement de point de vue.
Vous pouvez lancer le programme de la façon suivante :

```
./2Dcomp -s 1024 -k mandel -v seq
```

L'option `-s 1024` spécifie que l'on souhaite obtenir une image de 1024×1024 pixels.
Sans surprise, la version séquentielle `mandel_compute_seq` se trouve dans le fichier `src/mandel.c`. Jetez-y un oeil. Notez qu'on n'utilise plus qu'une seule image (`cur_img`), dont on écrase les pixels à chaque nouvelle itération.
Une version parcourant l'image en « tuiles » (`mandel_compute_tiled`) vous est également fournie. Pour l'essayer :

```
./2Dcomp -s 1024 -k mandel -v tiled
```

5.1 Travail demandé

Écrivez quatre versions parallèles de ces deux versions) :

ompb parallélisation de `mandel_compute_seq` en utilisant une politique de distribution statique (`b` pour block) ;

ompc_k parallélisation de `mandel_compute_seq` en utilisant une politique cyclique (`static, k`);

ompd_k parallélisation de `mandel_compute_seq` en utilisant une politique de distribution dynamique (`dynamic, k`);

omptiled parallélisation de `mandel_compute_tiled` en utilisant une politique de distribution dynamique des tuiles à l'aide de la directive `collapse`.

Déterminez les paramètres qui donnent de bons résultats « en moyenne » : valeur `k` pour `schedule(static, k)`, valeur de `GRAIN` pour les versions tuilées... Il est en effet possible de lancer le calcul dans des zones différentes de l'espace. Par défaut, vous utilisez la configuration 1 (cf les valeurs initiales des variables globales `leftX`, etc. dans le fichier `src/mandel.c`) dans laquelle les pixels coûteux en calcul apparaissent progressivement sur la droite de l'image. Mais si vous essayez la configuration 2, vous aurez alors affaire à une forte charge apparaissant depuis le bas de l'image.

5.2 Expériences à mener

Il s'agit de produire un graphique montrant l'accélération obtenues par vos versions parallèles en fonction du nombre de threads utilisés. On utilisera quelques centaines d'itérations.
Il n'est pas demandé de tracer une courbe pour chaque configuration de départ et chaque valeur possible de paramètre. Vous indiquerez simplement quels sont la configuration et les paramètres choisis pour chaque courbe.
Notez que, si en phase de mise au point de vos algorithmes vous pouvez accélérer le rendu en diminuant la taille de l'image (option `-s`), veillez bien à utiliser des images de taille significative lors de la production de vos courbes.
Examinez le script shell `script/lancer-expe.sh` qui vous aidera à automatiser le lancement de vos expériences. Si tout vous semble OK, lancez-le (cette étape pourra durer longtemps...)
Utilisez ensuite le script R situé dans le répertoire `script/` pour produire le graphique demandé en remplaçant `XXX` par le temps mesuré par vos soins pour la version séquentielle :

```
Rscript script/tracer-speedUp.R omp omp_d XXX
```