Programmation parallèle sur CPU multicoeurs
"Optimization using OpenMP"

A.Saadane

12 janvier 2021

TP 1^{ere} séance : Exemples d'applications

Exercice 1 : Création d'une région parallèle

1. En utilisant votre éditeur de texte favori, écrire un simple programme C qui :
   - crée une région parallèle (#pragma omp parallel) (nécessité d'inclure le fichier "omp.h"),
   - récupère l'identifiant de chaque thread dans la région parallèle (tid = omp_get_thread_num()),
   - fait afficher par chaque thread "Hello world" et son identifiant (tid).
2. Compiler ce programme en faisant appel à la librairie OpenMP (*gcc -o Hello Hello.c -fopenmp*)
3. Exécuter ce programme et noter le nombre de threads crées . Expliquer pourquoi.
4. Varier le nombre de threads (omp_set_num_threads(n)), Exécuter votre code plusieurs fois, observer à chaque fois la sortie et commenter.
5. Dans la région parallèle, faites afficher par le thread master (thread 0) le nombre total de threads. Déclarer le nombre de threads comme variable partagée et comme variable privée et observer la différence.

Exercice 2 : Boucles, partage de variables et ordonnancement

Le programme C suivant permet de définir et calculer la somme de 2 tableaux.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 100
#define CHUNKSIZE 10

void gen_ data(float *in1, float *in2, int length)
{
int i;
for (i=0; i < length; i++) {
in1[i] = i * 1.0;
in2[i] = i * 2.0;}
}
void add_ cpu(float *in1, float *in2, float *out, int length) {
int i;
for (i=0; i < length; i++) {
out[i] = in1[i] + in2[i];}
}

int main (int argc, char *argv[])
{
int i,chunk;
float a[N], b[N], c[N];
double starttime, stoptime, executiontime;
/* Génération des données*/
```

```
gen_data(a,b,N);
/* Calcul de la somme sans openmp et mesure du temps*/
starttime = omp_get_wtime();
add_cpu(a,b,c,N);
stoptime = omp_get_wtime();
executiontime = stoptime-startime;
return 0;
}
```

L'objectif de cet exercice est évidemment de paralléliser le traitement en répartissant le travail sur un certain nombre de threads. Pour cela :

- Commencer par écrire une fonction " void add_omp" avec pour arguments les 2 tableaux d'entrée, le tableau de sortie, la longueur des tableaux et une variable entière "chunk".
- Dans cette fonction, créer une région parallèle (#pragma omp parallel) avec déclaration des variables comme variables partagées. l'identifiant du thread (voir point suivant) sera déclaré comme variable privée.
- Une fois la région parallèle créée, commencer par récupérer l'identifiant du thread courant (tid = omp_get_thread_num()) et afficher le comme le thread qui commence le travail.
- Lancer ensuite l'ordonnancement de la boucle for (#pragma omp for) avec un ordonnancement statique et un CHUNKSIZE égal à "chunk".
- Faire afficher le thread et le coefficient out_omp[i] qu'il calcule.
- Dans le "main", fixer le nombre de threads à 4 (omp_set_num_threads(4)) et affecter CHUNSIZE à chunk.
- Appeler la fonction add_omp et calculer son temps d'exécution.

Une fois le codage terminé et la compilation réussie,

- exécuter plusieurs fois le programme et observer la sortie. Préciser quel thread traite quel chunk.
- Exécuter plusieurs fois le code et vérifier que l'ordonnancement statique est bien déterministe.
- Changer dans votre code source l'ordonnancement statique en un ordonnancement dynamique. Recompiler, exécuter à nouveau le programme et observer la différence.
- Vérifier que l'ordonnancement dynamique n'est pas déterministe (est-ce que les mêmes threads exécutent les mêmes chunks ?). Est-il possible qu'un thread n'ait rien à faire ?
- Fixer N = 500000 et CHUNKSIZE = 25000, supprimer tout ce qui n'est pas nécessaire au calcul de la somme (récupération du tid, printf) et comparer la performance (temps d'exécution) des fonctions add_cpu et add_omp. Conclure.

Exercice 3 : Produit scalaire - clause reduction

Le programme C suivant permet de définir et calculer le produit de 2 vecteurs.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 40

void gen_data_cpu(float *in1, float *in2, int length)
{
int i;
```

```
for (i=0; i < length; i++)
in1[i] = in2[i] = i * 1.0;
}
float prod_cpu(float *in1, float *in2, int length) {
int i;
float out;
for (i=0; i < length; i++) {
out = out + (in1[i] * in2[i]);}
return out;
}

int main (int argc, char *argv[])
{
int i;
float a[N], b[N], result_cpu=0.0;
gen_data_cpu(a,b,N);
result_cpu = prod_cpu(a,b,N);
}
```

L'objectif de cet exercice est d'illustrer l'intérêt de la clause "reduction" à l'intérieur d'une région parallèle et de quantifier la performance de OpenMP. Pour cela :

- Commencer par écrire la fonction "gen_data_omp" qui répartit la génération des données sur les différents threads. Utiliser la directive #pragma omp parallel for.
- En vous inspirant de la fonction "prod_cpu", écrire la fonction "prod_omp". Une façon de faire peut être de :
  - créer un région parallèle par la directive "#pragma omp parallel" pour créer des threads (on veillera dans ce cas à ce que les vecteurs d'entrée soit partagés). On peut récupérer et afficher le nombre de threads crées
  - répartir ensuite le travail effectué au sein de la boucle for sur les threads crées précédemment (#pragma omp for). Vous utiliserez pour cette dernière directive la clause "reduction (+ :out_omp) pour permettre à chaque thread de traiter un certaine nombre d'itérations et de mettre le résultat dans sa propre copie de out_omp. A la fin de la directive, les 4 copies des 4 threads sont additionnées pour le calcul du produit scalaire final (remarque : ces deux étapes peuvent se regrouper en une seule directive "#pragma omp parallel for").
  - afficher le résultat final pour vérification.

Une fois le codage terminé et la compilation réussie,

- exécuter le code et vérifier le bon calcul du produit scalaire.
- Commenter les résultats de la boucle for avec la clause reduction utilisée pour le calcul du produit scalaire.
- Comme pour l'exercice précédent, récupérer le nombre de threads et afficher le travail effectué par chaque thread. Le nombre d'itérations est-il le même pour tous les threads ?
- Quand un thread s'occupe de plus d'une itération, quel calcul effectue-t'il ?
- Les itérations traitées par le même thread sont-elles successives ?
- Relancer plusieurs fois le code. Est-ce que les mêmes threads traitent toujours les mêmes lignes ?
- Expliquer pourquoi à votre avis, la clause réduction est nécessaire.
- Supprimer cette clause de votre programme, recompiler, relancer plusieurs fois le code et commenter.

- Fixer N = 500000 et redéfinir les vecteurs d'entrée (in1[i]=in2[i]=i*1e-5), supprimer tout ce qui n'est pas nécessaire au calcul du produit scalaire (tid, nthreads, printf...) et comparer la performance (temps d'exécution) des fonctions "prod_cpu" et "prod_omp".

Exercice 4 : Produit matriciel

Le programme C suivant permet de définir et calculer le produit de 2 matrices.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

/* Utiliser d'abord des petites matrices pour tester

#define NRA 16 /* nombre lignes matrice A */
#define NCA 16 /* nombre colonne matrice A */
#define NCB 12 /* nombre colonne matrice B */

void gen_matA_cpu(double a[][NCA], int nra, int nca)
{
int i, j;
for (i=0; i<nra; i++)
for (j=0; j<nca; j++)
a[i][j]= (double) (i+j);
}
void gen_matB_cpu(double b[][NCB], int nrb, int ncb)
{
int i, j;
for (i=0; i<nrb; i++)
for (j=0; j<ncb; j++)
b[i][j]= (double) (i*j);
}
void init_result_cpu(double c[][NCB], int nra, int ncb)
{
int i, j;
for (i=0; i<nra; i++)
for (j=0; j<ncb; j++)
c[i][j]= 0.0;
}
void mat_mult_cpu(double a[][NCA], double b[][NCB], double c[][NCB], int nra, int nca, int ncb)
{
int i, j, k;
for (i=0; i<nra; i++)
for (j=0; j<ncb; j++)
for (k=0; k<nca; k++)
c[i][j] += a[i][k] * b[k][j];
}
```

```
int main (int argc, char *argv[])
{
int i, j;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
init_result_cpu(c, NRA, NCB);
gen_matA_cpu(a, NRA, NCA);
gen_matB_cpu(b, NCA, NCB);
mat_mult_cpu(a, b, c, NRA, NCA, NCB);
}
```

L'objectif de cet exercice est de gagner en performance en répartissant les différentes itérations sur les threads disponibles. Pour cela,

- commencer par réécrire les 4 fonctions en répartissant les itérations de chacune des fonctions sur les threads disponibles (utiliser pour chacune des fonctions la directive # pragma omp parallel for).
- Fixer un ordonnancement statique avec une taille de chunk à choisir.
- Vérifier que votre nouveau code est opérationnel en comparant vos résultats à ceux fournis par le code cpu.
- Augmenter maintenant la taille des matrices (à 256*256 par exemple) et comparer les temps d'exécution des traitements séquentiel (cpu) et parallèle (omp).
- Comment varie ce temps d'exécution en fonction de la taille du chunk.
- Choisir maintenant un ordonnancement dynamique, comparer les performances et conclure.

Pour cet exercice, vous pouvez également vous amuser à afficher le nombre de threads, le thread courant et le traitement qu'il effectue.

TP 2^{eme} séance : Traitement d'images avec OpenMP et OpenCV

Installation OpenCV

Le contexte de ce TD est celui du traitement d'images. Pour ce faire la librairie OpenCv est nécessaire. OpenCV est déjà installée sur les cartes fournies. Si tel n'est pas le cas, vous pouvez procéder à l'installation des 2 paquets essentiels moyennant la commande

sudo apt-get install libopencv-dev libcv-dev

La compilation d'un code écrit en C "prog.c" se fait par

$gcc -o prog prog.c 'pkg-config --cflags opencv --libs opencv' -lm -fopenmp (attention c'est ` et non ')

Utilisation de la directive "sections"

Exercice 1 : Calcul du négatif d'une image

L'objectif de cet exercice est de montrer comment le traitement d'image peut être accéléré moyennant OpenMP et l'utilisation en parallèle de plusieurs coeurs.

L'exercice proposé vise à calculer et à afficher le négatif d'une image. Le négatif d'une image est une opération usuelle en traitement d'image où le pixel de l'image de sortie I_s est le complément (à 255) de la valeur du pixel correspondant de l'image d'entrée I_e

$$I_s(x, y) = 255 - I_e(x, y)$$

Le code C suivant permet de calculer ce négatif.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <opencv/cv.h>
#include <opencv/highgui.h>

#define uchar unsigned char
#define imgsize_cpu 1024*1024*3

void negimage_cpu(uchar *array) {
int i;
for (i=0; i<imgsize_cpu; i++) array[i] = 255 - array[i];
}

int main (int argc, char *argv[])
{
double startime, stoptime, extime;
```

```
IplImage* img = NULL;
uchar *data;
const char* window_title = "Hello, OpenCV!";

// Opening the image file
if (argc < 2)
{
fprintf (stderr, "usage : %s IMAGE\n", argv[0]);
return EXIT_FAILURE;
}
img = cvLoadImage(argv[1], CV_LOAD_IMAGE_UNCHANGED);
if (img == NULL)
{
fprintf (stderr, "couldn't open image file : %s\n", argv[1]);
return EXIT_FAILURE;
}
data = (uchar *) img-> imageData;

// Getting the execution time
starttime=omp_get_wtime();
negimage_cpu (data);
stoptime=omp_get_wtime();
extime= stoptime-starttime;
printf("Total CPU execution time : %3.6f ms\n",extime*1000);

// Displaying the result
cvNamedWindow (window_title, CV_WINDOW_AUTOSIZE);
cvShowImage (window_title, img);
cvWaitKey(0);
cvDestroyAllWindows();
cvReleaseImage(&img);
return EXIT_SUCCESS;
}
```

Écrire et compiler ce programme. Relever le temps que met le CPU à calculer le négatif d'une image couleur de taille 1024x1024.

On cherche maintenant à améliorer ce temps d'exécution en utilisant la directive "sections" d'OpenMP. La directive "sections" permet l'exécution parallèle de différentes fonctions. L'idée est donc de répartir le calcul sur les 4 coeurs en demandant à chaque fonction de traiter une partie de l'image d'entrée. L'image elle-même est lue et affichée par les fonctions d'openCV. Pour assigner différents blocs d'opérations à chaque thread qui exécute une section, il faut donc

- commencer par définir la taille du bloc (de l'image) que chaque thread doit traiter. Disposant de 4 threads, l'idéal est donc de définir la taille d'un bloc comme étant le 1/4 de celle de l'image d'entrée (#define imgsize_omp 1024*256*3).
- Créer 4 fonctions de noms différents (negimage1, negimage2, negimage3 et negimage4 par exemple) effectuant le même calcul du négatif que la fonction negimage_cpu (la seule différence réside bien sûr dans la taille du bloc).
- Écrire maintenant la fonction "negimage_omp (uchar *array) qui commence par

- lancer la directive "parallel" pour répartir le travail à effectuer sur les 4 threads.
- Lancer ensuite la directive "sections" pour créer les sections.
- Créer la première section (#pragma omp section) et appeler la fonction negimage1 pour le calcul du négatif du premier bloc.
- Faire de même pour les 3 autres blocs (création de 3 sections et appel aux fonctions negimage2, negimage3 et negimage4).
- A la fin du traitement du 4eme bloc, penser à fermer la directive "'parallel" pour fusionner les résultats.

– Dans le main, appeler la fonction negimage_omp et calculer son temps d'exécution.

Après compilation, exécuter le programme, comparer les performances et commenter.

Analyser et tester le code avec et sans la directive "parallel".
Modifier la fonction negimage_omp pour n'utiliser qu'une seule fonction (negimage()) partagée entre les sections. Comment expliquez-vous le temps d'exécution.

Exercice 2 : Transformation colorimétrique (RGB/Niveaux de gris) et normalisation

Ce deuxième exercice vise à mettre en oeuvre la transformation colorimétrique permettant de passer d'une image couleur (3 composantes) à une image en niveaux de gris qui sera ensuite normalisée. La transformation colorimétrique considérée est celle où l'image monochrome de sortie $I_{bw}(i,j)$ est donnée par

$$I_{bw}(i,j) = 0.3 * I_r(i,j) + 0.59 * I_v(i,j) + 0.11 * I_b(i,j)$$

où $I_r(i,j), I_v(i,j), I_b(i,j)$ représentent respectivement les composantes rouge, verte et bleue de l'image couleur d'entrée. La normalisation est un processus de recalage de l'image utilisé pour fixer les niveaux de gris dans un intervalle donné. Dans cette exercice, les niveaux de gris sont modifiés de telle sorte qu'ils aient une moyenne nulle et un écart type égale à l'unité (courant dans l'analyse d'images). Si l'on appelle ν la valeur moyenne de l'image d'entrée $I_{bw}(i,j)$ et σ son écart type, alors l'image de sortie normalisée $I_{bw}(i,j)$ est donnée par :

$$I_{bwn}(i,j) = \frac{I_{bw}(i,j) - \nu}{\sigma}$$

Travail demandé

1. Le fichier "grey_norm_to_be_completed.c" fournit le code C correspondant aux opérations de transformation colorimétrique et normalisation. Rajouter l'estimation du temps d'exécution, compiler et évaluer la performance CPU.
2. A l'instar de ce qui a été fait dans l'exercice 1, écrire la nouvelle fonction bwimage_omp (directives parallel et sections)
3. Paralléliser le calcul de la moyenne, de l'écart type et de la normalisation (pragma omp parallel for) en jouant sur toutes les directives déjà utilisées (shared, private, firstprivate, reduction et schedule).
4. Évaluer les performances de votre nouveau code et conclure.

A l'instar de ce qui a été fait dans l'exercice 1, écrire la nouvelle fonction bwimage_omp

Rotation d'image