Programmation Parallèle OpenMP

RAPPORT TP3-4

EL RIFAI Omar

TPA11

Analyse fonctionnelle :

- ## ColorImageToGrey :

```
/*****************************************************
 * Transforms a color image PPM to grey PGM
 *****************************************************/
grey_image_type* colorImageToGrey(color_image_type *colorImage){
  double t, start, stop;
  start = omp_get_wtime();
  int width=colorImage->width, height=colorImage->height;
  grey_image_type *greyImage=createGreyImage(width, height);
  int r, g, b;
#ifdef PARALLEL
#pragma omp parallel for private(r, g, b) num_threads(PARALLEL)
#endif
  for (int i = 0; i < width * height; i++){
    r=colorImage->pixels[i].r;
    g=colorImage->pixels[i].g;
    b=colorImage->pixels[i].b;
    greyImage->pixels[i] = (299 * r +587 * g + 114 * b) / 1000;
  }
  stop = omp_get_wtime();
  t = stop - start;
  printf("TEMPS-colorToGrey: %lf\n", t);
  return greyImage;
}
```

***Figure 1*: ColorImageToGrey**

Analyse:

On remarque que cette fonction fait un parcours de chaque pixel de l'image pour faire des affectations de valeur. Les opérations à chaque itération sont indépendants. Normalement, le nombre d'itérations augmente avec l'augmentation de la qualité/taille de l'image.

Conclusion :

Le parallélisation sera plus efficace quand les images sont de grandes tailles, et moins efficace quand elles sont de petites tailles.

- ## GreyImageToColor :

Analyse:

On remarque que cette fonction fait un parcours de chaque pixel de l'image pour faire des affectations de valeur. Les opérations à chaque itération sont indépendants. Normalement, le nombre d'itérations augmente avec l'augmentation de la qualité/taille de l'image.

Conclusion :

La parallélisation sera plus efficace quand les images sont de grandes tailles, et moins efficace quand elles sont de petites tailles.

```
/*****************************************************
 * La fonction d'embossage
 *****************************************************/
grey_image_type* greyImageToColor(grey_image_type* greyImage){
  double t, start, stop;
  start = omp_get_wtime();
  int width=greyImage->width, height=greyImage->height;
  grey_image_type* colorImage=createGreyImage(width, height);
#ifdef PARALLEL
#pragma omp parallel for num_threads(PARALLEL)
#endif
  for (int i = 0; i < width*height; i++){
    if(i < width || i % width == 0 || (i+1)% width ==0|| (height-1)*width <= i ){
      colorImage->pixels[i]=greyImage->pixels[i];
    }else{
      colorImage->pixels[i]= greyPixelToColor(greyImage->pixels, i, width, height);
    }
  }
  stop=omp_get_wtime();
  t = stop - start;
  printf("TEMPS-greyTocolor: %lf\n", t);
  return colorImage;
}
```

***Figure 2*: greyImageToColor**

- ## Transform :

Analyse:

On remarque que dans cette fonction on utilise 3 boucle « for ». La première et la dernière boucle font autant d'itérations que le nombre de pixels dans une image. Alors que la deuxième ne fait que 256 itérations sauf qu'à chacune itérations elle parcours i fois un tableau. Les opérations qui sont faites à chaque itérations de la 2ème et 3ème boucle sont complètement indépendantes l'une de l'autre. Par contre, dans la 1ère boucle on incrémente les valeurs dans un tableau sans aucun ordre précis.

***Figure 3*: transform (séquentielle)**

Conclusion :

On gagne réellement une accélération importante par la parallélisation de la première et troisième boucle. Sachant que l'instruction dans la première boucle doit être atomique. Théoriquement, si on parallélise la deuxième boucle on gagne une accélération mais réellement ce n'est pas le cas car le nombre d' itérations par chaque threads va augmenter avec « i ». Donc, l'utilisation d'un schedule dynamique/guided pour la deuxième boucle est réellement plus efficace.

Expérimentations :

- ## ColorImageToGrey :

```
#pragma omp parallel for private(r, g, b) num_threads(PARALLEL)
```

On place le **pragma** de ci-dessus juste avant la boucle for (voir *Figure 1: ColorImageToGrey*). Les variables privées sont **r, g et b** car c'est des variable utilisées localement pour calculer la valeur d'un pixel. Le **num_threads()** est utilisée pour qu'on précise le nombre de threads qu'on veut créer. Le nombre de threads évident à choisir est **2** car ma machine peut exécuter jusqu'à 2 threads en parallèle. Voyons les temps d'exécution ci-dessous. On remarque que l'accélération réelle se rapproche vers l'accélération théorique quand la taille de l'image (i.e nombre d' itérations dans la boucle for augmente suffisamment)

Temps en (s)	Image0.ppm(535Ko)	Image2.ppm(75.1Mo)
Parallèle (2 Threads)	0.000312	0.020967
Séquentiel	0.000366	0.041709

Acc. Théorique / 2	1.99	1.99
Acc. Réelle / 2	1.17	1.98

- ## GreyImageToColor :

```
#pragma omp parallel for num_threads(PARALLEL)
```

On place le **pragma** de ci-dessus juste avant la boucle for (voir *Figure 2: greyImageToColor*). Les variables privées sont **r, g et b** car c'est des variable utilisées localement pour calculer la valeur d'un pixel. Le **num_threads()** est utilisée pour qu'on précise le nombre de threads qu'on veut créer . Le nombre de threads évident à choisir est **2** car ma machine peut exécuter jusqu'à 2 threads en parallèle. Voyons les temps d'exécution ci-dessous. On remarque que l'accélération réelle est très proche de l'accélération théorique dans les deux cas indépendamment de la taille de l'image, car les opérations faites à l'intérieure de la boucle sont importantes, donc même sur un nombre d'itérations petit l'accélération est importante

Temps en (s)	Image0.ppm(535Ko)	Image2.ppm(75.1Mo)
Parallèle (2 Threads)	0.000573	0.053172
Séquentiel	0.001121	0.103714
Acc. Théorique / 2	1.96	1.95
Acc. Réelle / 2	1.99	1.99

- ## Transform :

- On place le **pragma** juste avant la boucle for (voir Figure 4: Transform-1ere boucle for). La variable privée est **pixel** car c'est une variable utilisée localement comme un indice pour accéder au tableau **H[]**. Le **num_threads()** est utilisée pour qu'on précise le nombre de threads qu'on veut créer . Le nombre de threads évident à choisir est **2** car ma machine peut exécuter jusqu'à 2 threads en parallèle. Par contre, la variable privée **pixel** peut avoir une valeur identique d'une itération à une autre. Alors l'accès et l'incrémentation à une case de tableau **H[]** sont susceptibles d'être fait en même temps par plusieurs threads car ces opérations ne sont pas **atomiques**. Donc il faut les mettre dans **une section critique** (voir Figure 4: Transform-1ere boucle for).

  Mais après quelques mesures de temps d'exécution de cette fonction, j'ai remarqué que le fait qu'il y a **une section critique** augmente de **2x** le temps d'exécution. Donc pour augmenter l'accélération réelle, il vaut mieux ne pas paralléliser cette boucle.

```
grey_image_type* transform(grey_image_type* greyImage){
  double t, start, stop;
  start = omp_get_wtime();
  int width=greyImage->width, height=greyImage->height;
  grey_image_type* contrastImage=createGreyImage(width, height);
  int *H, *C;
  H=calloc(NB_PIXELS, sizeof(int));//POUR S'ASSURER QUE LES CASES DE MEMOIRES RESERVEES SONT INITIALISEES A ZERO
  C=calloc(NB_PIXELS, sizeof(int));
  unsigned char pixel=0;
#ifdef PARALLEL
#pragma omp parallel for private(pixel) num_threads(PARALLEL)
#endif
  for (int i = 0; i < width*height; i++){
    pixel=greyImage->pixels[i];
#ifdef PARALLEL
//CET INSTRUCTIONS DOIT ETRE ATOMIQUE, SINON PLUSIEURS ACCES ET MODIFICATION PEUVENT ETRE FAIRE EN MEME TEMPS
#pragma omp critical
{
#endif
    H[pixel]++;
#ifdef PARALLEL
    }
#endif
  }
```

Figure 4: ***Transform-1ere boucle for***

- On place le deuxième **pragma** juste avant la deuxième boucle for (voir le code ligne 188).Par contre, cette boucle ne fait que **256** itérations dont chacune fait « **i** » itérations. Alors le nombre d'opérations faites à chaque itérations augmente avec « **i** ». Donc les threads qui vont exécuter les dernières itérations vont mettre le plus du temps à finir alors que ceux qui vont faire les toutes premières itérations finiront plus tôt.
  Une solution est de mettre un **schedule(dynamic, 50)** ou guided, pour ne laisser aucune thread finir et se mettre en attente pour les autres. Le nombre **50** représente le nombre d'itérations que chaque thread va exécuter avant de voir s'il en reste d'autre. Cette solution est théoriquement efficace, mais réellement (comme ma machine ne fais que 2 threads en parallèle) ne diminue pas le temps d'exécution(0 effets).
  Voici les temps d'exécution de la fonction transform avec différents cas.

Temps en (s)	image0	image2	Acc. Réelle image2
(#2,3 **for**) parallélisées	0.000553	0.041679	1.66
(#1,2,3 **for**) parallélisées	0.002300	0.179062	0.39(negatif)
(#1,3 **for**) parallélisées	0.002555	0.160441	0.43(negatif)
(#3 **for**) parallélisée	0.000516	0.064092	1.08
Séquentielle	0.000553	0.069203	----------

- Pour la derniere boucle idem GreyImageToColor