# OpenMP TP3 Rapport

Code « propre » avec headers disponible à
https://github.com/Rhhet/omp_tp3/tree/omp_tp3_win

## Contenu du code source :

- `main()` : commence par tester si l'image que l'on souhaite transformer n'a pas déjà été transformée en niveaux de gris (utilisation de `fopen()`) ;
  Si l'image a déjà été transformée, on utilise la fonction `loadGreyImage()` qui reprend le code de `loadColorImage()` et load en mémoire (sur le *heap*) l'image en niveaux de gris.
  Sinon, l'image est transformée en niveaux de gris via la fonction `color_to_grey()`. Cette distinction est due au fait qu'il est inutile de retransformer l'image si cela a déjà été fait, sachant que la fonction `loadColorImage()` est la plus lourde (la plus longue à éxécuter, environ 2 sec pour l'image 2) du programme, alors que `loadGreyImage()` demande en général deux fois moins de temps.
  Les deux fonctions renvoient un pointeur sur l'image effectivement transformée en niveaux de gris qui est ensuite passé en paramètres des fonctions d'embossage (`emboss_pgm()`) et de contraste (`contrast_pgm()`).
  La fonction `clean_gimg()` est appelée pour libérer le heap de l'image en niveau de gris.
- `color_to_grey()` : la fonction commence par loader l'image en couleur en mémoire (sur le heap) via `loadColorImage()` puis crée une image en niveaux de gris vide via `createGreyImage()`. Après transformation, l'image est sauvegarder sur le heap avec `saveGreyImage()`.
- `emboss_pgm()` : prend en paramètre un pointeur vers une image en niveau de gris. Après création d'un nouvelle image en niveaux de gris vide, la fonction transforme chaque pixels en appelant la fonction intermédiaire `emboss_pixel()` décrite plus loin, qui calcule le nouveau pixel.
- `contrast_pgm()` : transforme l'image en niveau de gris pointée par le pointeur passé en argument en sa version contrastée. Utilisation de `calloc()` à la place de `malloc()` afin d'assurer que les cases mémoire allouées sont initialisées à 0 (cf code) lors du calcul des histogrammes.

  La fonction `emboss_pgm()` utilise la fonction intermédiaire `emboss_pixel()` qui, étant donné un pixel d'indice i, calcule le nouveau pixel

à partir des coefficients de pondération. La fonction `emboss_pixel()` commence par extraire la matrice 3 * 3 autour du pixel via la fonction `extract_mat()` puis calcule le produit scalaire entre cette matrice (vue comme vecteur) et la matrice des coefficients de pondération en utilisant la fonction `scalar_prod()` codée à cet effet.

NOTE : les fonctions de transformation utilisent les fonctions `clean_gimg()` et `clean_cimg()` afin de libérer le heap des images qui ne sont plus utilisées (c'est notamment le cas de l'image en couleur, qui une fois chargée en mémoire n'est utilisée que pour créer l'image en niveaux de gris, elle n'est plus utilisée par la suite et va encombrer le heap).

## Paralélisation des fonctions :

- fonction `color_to_grey()` : on ne parallélise ici uniquement la boucle `for` étant donné que le reste du code ne demande quasiment aucun calcul (simples affectations et allocations de mémoire) ; sans compter l'appel aux fonctions `loadColorImage()`, et `saveGreyImage()` traitées séparément. La boucle est parallélisée par un `omp parallel for` avec spécification du nombre de threads (la variable de boucle est initialisée dans la boucle et est donc automatiquement privée pour chaque thread). Ici aucun dépendance n'empêche la parallélisation, les threads lisent simplement le même tableau `cpixels` sans le modifier et écrivent dans le tableau `gpixels` sans dépendance (les cases sont indépendantes).

- fonction `emboss_pgm()` : Les deux premières boucles `for` ne sont pas parallélisées car elles n'effectuent que très peu d'itérations (largeur de l'image itérations) comparé à la troisième boucle, il n'y aura donc aucun réel gain à paralléliser ces boucles. La troisième boucle est parallélisée de la même façon que pour la fonction précédente, il n'y a aucune dépendance ici, y compris dans la fonction intermédiaire `emboss_pixel()`.

- fonction `contrast_pgm()` : On crée dans cette fonction une team de threads au préalable qui se partage la première boucle `for` pour le calcul de l'histogramme, puis seul un thread exécute le calcul de l'histogramme cumulé via une clause `single`. Les threads se rejoignent ensuite pour la dernière boucle `for` afin de calculer les nouveaux pixels.

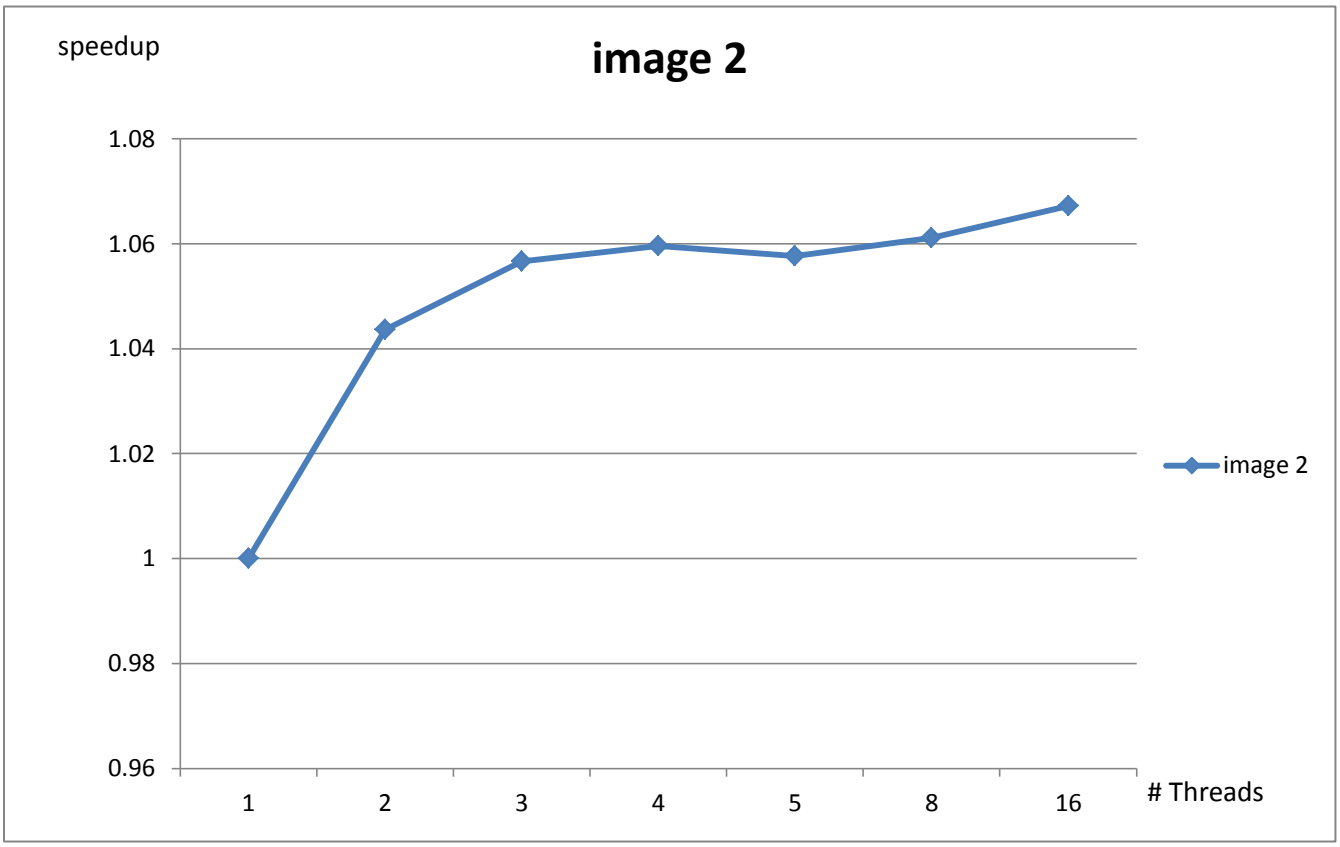
NOTE : Les fonctions les plus lourdes du programme sont visiblement les fonctions de traitement des images (lecture et écriture). Elles utilisent, après calculs, environ 90% du temps d'exécution du programme. Il semblerait logique de tenter de les paralléliser. Je me suis heurté au fait que les fonctions `fprintf()` et `fscanf()` sont thread-safe et visiblement dotées d'un mutex automatique, les streams sont protégés par un lock. Je ne sais pas comment lire/écrire dans un fichier (unique) dans un environnement multithread. Ceci n'a pas été vu en cours, les seules sources sur ce sujet traitent en C++. (Peut être en ouvrant un stream différent pour chaque thread, pointant sur le même fichier ? auquel cas nécessaire de prédécouper le fichier au préalable, sinon les threads écrivent/lisent au mauvais endroit…).

## Temps d'exécution et speedup :

- On utilise la fonction `omp_get_wtime()` sur les morceaux de code importants (en l'occurrence, les boucles parallélisées, car le reste du code est négligeable) pour récupérer le temps écoulé.
- En page suivante : le tableau récapitulatif donnant les temps pour chaque fonction lors du traitement de l'image 2 (la plus significative). Les calculs de temps sont fait en supposant que l'image n'a pas déjà été transformée en niveaux de gris (la colonne `load_grey()` est ignorée). La fonction `save_grey()` est appelée 2 fois. Le speedup est donnée en dernière colonne.
  Est donné également la courbe du speedup en fonction du nombre de threads.

- Analyse : Etant donné que le programme passe presque 90% de son temps à exécuter les fonctions de lecture/écriture des images, le speedup est très petit (la portion parallélisable est vraiment petite ici). Ce speedup serait probablement bien plus intéressant s'il était possible de paralléliser ces fonctions.
  Les calculs ont été effectués avec un Intel core I5 2500K (4 cœurs) 4GHz, ce qui explique pourquoi le speedup devient moindre à partir de 5 cœurs. Le gain est significatif (relativement à la proportion de code parallélisé) avec 2, 3 et 4 cœurs par rapport au single-core.
  On pourrait s'intéresser à la loi d'Amdhal pour trouver le speedup maximal théorique. En utilisant une proportion de 92% de code séquentiel, on obtient des résultats plutôt cohérents (speedup théorique de 1.06 pour 4 cœurs par exemple).

| functions / nb of threads | color -> grey | load color | load grey | save grey | emboss | contrast | **exec time (sec)** | speedup |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.047 | 2.05 | 0.88 | 1 | 0.36 | 0.04 | 4.497 | 1 |
| 2 | 0.052 | 2.05 | 0.88 | 1 | 0.18 | 0.027 | 4.309 | 1.043629612 |
| 3 | 0.052 | 2.05 | 0.88 | 1 | 0.13 | 0.024 | 4.256 | 1.05662594 |
| 4 | 0.051 | 2.05 | 0.88 | 1 | 0.12 | 0.023 | 4.244 | 1.059613572 |
| 5 | 0.051 | 2.05 | 0.88 | 1 | 0.125 | 0.0259 | 4.2519 | 1.057644818 |
| 8 | 0.054 | 2.05 | 0.88 | 1 | 0.11 | 0.024 | 4.238 | 1.061113733 |
| 16 | 0.051 | 2.05 | 0.88 | 1 | 0.09 | 0.023 | 4.214 | 1.067157095 |




**Grandpierre Téri**

Groupe A1.1