# LSINF1252 : Systèmes informatiques 1, Projet 2 : Application de filtres sur des images Présentation générale de la solution et choix architecturaux

de Mol Maxime, de Bodt Cyril, Groupe 1

19 avril 2013

Pour atteindre les objectifs de ce deuxième projet du cours *Systèmes informatiques 1*, portant sur le traitement parallélisé d'images, nous avons utilisé un système du type producteur-consommateur. Comment ? Un filtre a un certain nombre de threads permis. Chacun des threads d'un filtre x ira prendre une image, ou un morceau d'image, dans un buffer qui contiendra l'ensemble des images, ou morceaux d'images, devant encore être filtrés par x. Ensuite, ayant filtré cette image ou ce morceau d'image par le filtre x, le thread ira la (ou le) placer dans un buffer suivant, contenant l'ensemble des images, ou morceaux d'images, ayant été filtrés par x.

Ainsi, nous pouvons observé que nous avons un buffer suivant chaque filtre (sinon, nous ne pourrions récupérer les images ou morceaux d'images filtrés) et un premier buffer précédant le premier filtre, et qui contient toutes les images, ou morceaux d'images, de départ. L'adoption de ce système, combiné ave l'algorythme du producteur-consommateur, permettra de paralléliser l'application successive des filtres sur les différentes images.

Ayant défini le schéma globale de résolution, nous devons préciser un petit point : nous avons pris soin de spécifier, ci-dessus, qu'un thread d'un filtre x pouvait manipuler une image *ou un morceau d'image*. Quelle différence cela fait-il ? Dans le cas où chaque thread ne manie qu'une image, l'implémentation de la solution est plus simple mais il y aura une perte d'efficacité certaine, si il y a plus de threads accessibles que d'images. Dans le cas où chaque thread manipule un morceau d'image, le code sera d'avantage parallélisé, si l'on découpe les images en suffisamment de morceau pour s'assurer que chacun des threads ait toujours du travail. Ayant décider de paralléliser le plus possible le code, nous avons opté pour la deuxième solution et traiter des morceaux d'images. Ces morceaux d'images sont en réalité des paquets de lignes, au nombre du nombre maximale de thread permises pour un filtre. A présent, décrivons l'architecture de la solution.

### La fonction *main*

Après le lancement du programme, la première étape est de récupérer les arguments qui lui ont été donné. Ceci fait, nous pouvons commencer à initialiser les différentes structures de données utilisées : les fonctions *initializeStatic*, *initializeBlur* et *initFilters* sont prévues à cet effet. Notons que ces fonctions sont situées dans le fichier *threadpool.c* et servent à initialiser des variables static globales dans ce fichier [1]. Expliquez la raison d'être de toutes ces variables ici serait laborieux et c'est pourquoi, au fur et à mesure de notre expédition dans le code, nous détaillerons leur raison d'être.

La fonction *main* a donc pour fonction de lancer le programme, d'initialiser les variables et, une fois le traitement effectué, de sauvegarder les images dans le nouveau dossier, en libérant de la mémoire les structures allouées. Le traitement des images est, quant à lui, réalisé dans une autre fonction, *image_treatement*. Cette fonction est située dans le fichier *threadpool.c*, qui regroupe l'ensemble des fonctions nécessaires à la parallélisation du code.

1. Nous avons jugé plus élégant de travailler avec des variables globales plutôt qu'avec des arguments supplémentaires dans chacune de nos fonctions.

***image_treatement***

A nouveau, cette fonction débute par une phase d'initialisation des mutex et autres structures de données et se terminent en les libérant. A côté de ces détails d'implémentation, notons que la fonction lance les threads vers la fonction *productor*, pour que le travail du producteur-consommateur puisse commencer. Le nombre de threads par est connu grâce au tableau *nthread*, initialisé dans la *main* et qui contient le nombre de threads permises par filtres (un filtre étant représenté par un nombre, le premier filtre correspondant au 0, le deuxième au 1, ...). Avant de passer au producteur, notons que les arguments donnés aux threads sont bien spécifiques : nous les détaillerons plus tard.

**La fonction *productor***

Nous avons mentionné que nous avons décidé de partionner nos images en plusieurs morceaux, afin de davantage parralléliser le code, et d'utiliser des buffer entre les paires successives de filtres par les stocker. Dès lors, nous pouvons aisément comprendre que le filtre d'indice x va consommer le buffer x et produire dans le buffer x+1. Le buffer d'indice 0 sera celui où les morceaux dimages initiales seront traités et celui d'indice égale au nombre de filtre contiendra les morceaux d'images complètement traités. Tous ces buffers sont contenus dans une matrice, *mat_buf*, dont les éléments, les pointeurs vers les elem_buf, sont des pointeurs vers des structures que nous avons définies dans le fichier *threadpool.h*. Ces structures contiennent simplement une référence vers une image, des lignes de début et de fin, une référence vers une image read-only, que nous utiliserons dans le cas du flou gaussien, et un indice d'image (les images se sont vus attribuées des indices, qui vaut 0 pour la première image, 1 pour la suivante, ..., pour faciliter l'emploi des tableaux et matrices dont le nombre de colonnes ou de lignes est égal au nombre d'images).

Ayant défini défini *mat_buf*, il nous faut maintenant faire traverser les différents filtres par les morceaux d'images. En rentrant dans la fonction *productor* avec un argument qui indique le numéro du filtre, s'il s'agit d'un flou gaussien ou s'il précède le flou gaussien [2], les threads arrivent dans une boucle qui semblent infinie, mais qui se terminent lorsqu'il y a eu autant de passage dans la boucle que de nombre de morceaux d'images (la variable NPACK indique le nombre de morceaux d'images au total, toutes les images comprises), grâce à la variable to_pass, protégée par son mutex. Une fois cette étape franchie, il y a deux cas : soit le filtre courant est un flou gaussien, soit il ne l'est pas.

Analysons le cas le plus facile en premier lieu : la clause else, le filtre courant n'est pas un flou gaussien. La première étape est de consommer un paquet d'image du buffer précédant, se situant à la ligne de *mat_buf* de même numéro que celui du filtre. Pour cela, on utilise la fonction *consommator*, en lui spécifiant qu'on ne recherche pas un morceau d'une image particulière (argument -1). Cette fonction fait d'abord un *wait* sur la sémaphore *full* associée au filtre courant, pour ensuite *locker* le mutex associé à ce filtre, respectant ainsi l'algorythme du consommateur. Nous prenons ensuite un élément non-vide de la ligne courante de *mat_buf*, mettons la case correspondante à l'élément choisi à *NULL* pour enfin renvoyer un pointeur vers celui-ci, après le *unlock* sur le mutex et le

2. Ces deux valeurs valent 0 si le filtre n'est pas un flou gaussien (resp. ne précède pas un flou gaussien), 1 si c'est le premier flou gaussien (resp. précède le premier flou gaussien), 2 si c'est le deuxième, ...

post sur la sémaphore *empty* associée au filtre précédant. Notons que tous les *elem_buf* ont été stockés sur le tas.

Ayant un *elem_buf* à traiter, le producteur peut appliquer dessus la fonction *apply_filter* du fichier *filters.c*, fonction qui prend un pointeur vers un morceau d'image et un numéro et qui applique sur ce morceau d'image le filtre d'indice correspondant au numéro. Ayant appliqué le filtre, le producteur applique un wait sur la sémaphore *empty* correspondant au buffer suivant, pour *locker* le mutex associé au nouveau buffer, y insérer l'élément, faire le *unlock* et le post sur la sémaphore *full* du buffer suivant, respectant ainsi l'algorythme du producteur. Une dernière chose à effectuer concerne le cas où le filtre courant précède un flou gaussien (parg->beforeblur !=0). Nous savons que, pour appliquer un flou gaussien sur un morceau d'image, nous devrons avoir une copie de l'image sur laquelle tous les filtres précédant ont été appliqués. Dès lors, pour lancer la copie, il faut savoir si tous les morceaux de l'image sont bien passés par le filtre précdant. Pour cela, si notre filtre précède un flou gaussien, on incrémente un élément de *mat_blur*, une matrice avec autant de ligne que le nombre de flou gaussien et autant de colonne que d'images. Quel élément ? Celui dont la ligne correspond à parg->beforeblur - 1 [3] et dont la colonne correspond à l'indice de l'image à laquelle appartient le morceau d'image traité. Enfin, dans le cas où tous les morceaux d'une image ont été traités, nous effectuons un post sur la sémaphore dédié au filtre de flou gaussien qui va suivre. Cette sémaphore permet d'implémenter le problème du rendez-vous dans le cas du filtre blur, que nous expliquerons ci-dessous.

Ayant traité le cas du filtre qui n'est pas un flou gaussien, revenons au début de la boucle *while* pour voir ce qu'il en est lorsque le filtre courant est un flou gaussien. Le principe est le même, avec quelques petits ajouts. En effet, au lieu de directement appeler le consommateur, il faut d'abord vérifier s'il est une image dont tous les morceaux ont été traités par le filtre précdant. Pour faire cela, nous avons implémenter une solution sous la forme d'une barrière. Nous avons créé un tableau *can_copi* de sémaphores, avec autant d'éléments que de filtres de flou gaussien. Elles sont toutes initialisée à 0, sauf si le premier filtre est un flou gaussien, auquel cas la première ligne de *can_copi* contiendra des sémaphores initialisées au nombre d'images. Une fois dans le producteur dans le cas du flou gaussien, nous effectuons un *wait* sur la sémaphore associée au flou gaussien courant. Il n'y aura que deux manières de passer ce *wait* : soit tous les morceaux d'une image seront passés par le filtre précédant, auquel cas un post aura été effectué sur la sémaphore, soit le flou gaussien courant est le premier filtre, auquel cas la sémaphore n'est pas initialisée à 0. Ceci fait, il faut trouver l'image dont les morceaux sont prêts à passer par le flou gaussien. C'est le rôle de *check_lmat_blur*, qui va parcourir la ligne courante de *mat_blur* pour trouver l'image pouvant passer par le flou gaussien et renvoyer le numéro qui lui correspond. Cette recherche de l'image valide est enfermée dans un mutex, *can_i_take_c_mutex*, pour éviter des résultats incohérents. En effet, il ne faut pas qu'une image ait été sélectionnée plus de fois que $\frac{NPACK}{NIMAGE}$. Ce pourrait arriver s'il ne restait, pour une image, qu'un morceau à traiter, qu'une thread trouvait cette image via le *check_lmat_blur*, mais qu'elle soit directement bloquée après la recherche de sorte qu'une autre thread pourrait également se voir attribué ce numéro d'image par *check_lmat_blur*. Alors, une des deux threads resterait à jamais bloquée sur la sémaphore *wait* du consom-

3. parg->beforeblur valant 1 pour le premier filtre, il faut faire moins 1 pour être à la première ligne de *mat_blur*

mateur et le programme, attendant la fin de chacune des threads, ne se finirait pas. Dès lors, *check_lmat_blur*, si elle trouve un numéro d'image, incrmémente un élément de la matrice *can_i_take_c* qui correspond au couple (filtre gaussien ; image) courant. Pour empêcher le fait qu'il puisse y avoir un thread qui modifie *can_i_take_c* et un autre que l'utilise dans la recherche de *check_lmat_blur*, on enferme l'appel à *check_lmat_blur* dans un mutex, avec le *can_i_take_c_mutex*. On pourrait se demander comment deux threads pourraient se retrouver dans le *check_lmat_blur*, étant donné la sémaphore ? Eh bien, prenons le cas où la sémaphore est initialisée à zéro. Imaginons que deux images soit entièrement passées par le filtre précédant ; ceci aura comme conséquence que deux post seront effectués sur notre sémaphore du flou gaussien courant. Deux threads pourront donc passer, ce qui justifie le *can_i_take_c_mutex*. Enfin, on effectue un post sur notre sémaphore une fois la recherche effectuée, si le nombre de paquet de lignes de l'image courante qui ont été pris en charge par un thread est inférieur au nombre de paquet de ligne en lequel a été divisé l'image.

Ayant trouvé une image valide, la prochaine étape est de voir s'il faut la copier. C'est le rôle de *copi_is_done*. Si l'élément correspondant à la paire flou gaussien et image courante est nul, il faut faire la copie. Pour la faire, on utilise la fonction *image_copi*, qui elle-même utilise *memcpy*, et qui affecte à tous les *read-only* des morceaux de l'image choisie un pointeur vers la copie. Le reste du code effectue la même chose que dans le cas du filtre qui n'est pas un flou gaussien, à ceci près qu'il faut détruire la copie de l'image grâce à *image_copi_destroy* une fois que les $\frac{NPACK}{NIMAGE}$ morceaux d'images sont passés par le flou gaussien. Pour savoir s'il faut détruire la copie de l'image, on incrémente la variable *choosen_c* après avoir insérer le morceau d'image dans le buffer suivant, en la protégant avec un mutex qui est *choosen_c_mutex*. Après cette incrémentation, on peut vérifier si l'élément de *choosen_c* qui correspond au flou gaussien courant et à l'image courante vaut $\frac{NPACK}{NIMAGE}$ et si tel est le cas, on détruit la copie. Précisons pour finir qu'il faut *locker* le mutex *choosen_c_mutex* lorsqu'on vérifie sa valeur, afin qu'il n'y ait pas une thread qui modifie *choosen_c* pendant qu'une autre la lit. Ceci conclut la description globale de la solution.

### Problèmes rencontrés

Souhaitant adopté la solution la plus efficace, le principal problème rencontré fut de ne pas se perdre dans la complexité de cette solution.

En guise de remarque, notons que notre programme présente un très faible risque de buffer overflow, par exemple si un utilisateur entre un nom de fichier vraiment très très long. Aussi, si une erreur survient dans le programme, à cause d'une fonction comme *malloc*, *pthread_create*, ..., la fonction *error* est appelée et libère toutes les structures allouées sur le tas, avant d'arrêter le programme. Enfin, notons que notre solution gagnera en efficacité par rapport aux programmes séquentiels lorsque le nombre d'image et de filtre à appliquer sera grand. Alors, même si un flou gaussien est demandé, le temps mis par la copie sera presque insignifiant par rapport à celui gagné par la parallélisation du code. Cependant, si l'on imagine un appel du programme avec seulement trois filtres de flou gaussien et peu d'images, notre solution ne sera pas plus rapide, voire même plus lente qu'un programme séquentiel, à cause de la copie des images entre chaque filtre.