Prénom :
Nom :
Groupe TD :
Date :

Optimisations et Algorithmes Linéaires

M4105C

François Merciol

UBS - IUT

DUT Informatique – 2nde année

Vannes – 2018/2019

Support de cours **étudiant**

http://m4105.merciol.fr/

(page gauche vide)

Table des matières

Table des figures

Liste des tableaux

Listings

Liste des Algorithmes

1 Présentation du module

Ce module optionnel M4105C de l'IUT de Vannes traite de l'optimisation algorithmique. Il montre comment il est possible de gagner considérablement en temps d'exécution par des choix stratégiques. Il aborde des algorithmes en complexité linéaire, voire constante.

Il s'appuie sur un certain nombre de publications scientifiques récentes [1] [2] (novembre 2017), qui touchent la construction d'arbres en temps linéaire.

Ce module d'enseignement s'étale sur 7 semaines, à raison d'un créneau par semaine devant ordinateurs. Il sera donc constitué de 7 Travaux Dirigés (TD) en salle machine.

Il y a 2 phases de TP :

- TP1 : calcul pseudo-parallèle et effet sur la gestion de cache mémoire
- TP[2-7] : analyse d'images par représentation hiérarchique (sous forme d'arbres)

L'évaluation du module se fonde sur une note en fin de période. Elle porte sur le rendu d'un programme d'analyse d'images rédigé sur l'ensemble de la période.

Au delà de l'aspect pédagogique, ce module a pour ambition de vous montrer l'intérêt de la recherche contemporaine sur les outils que vous serez amenés à utiliser. Et pourquoi pas susciter des vocations...

1.1 Le rendu

Le résultat que vous devez rendre s'appuie sur la structure déjà fournie comme squelette et qui est disponible sur le site `http://m4105.merciol.fr/`. Pour identifier de manière fiable votre production, cette dernière doit inclure les noms des participants de votre binôme. Prenons comme hypothèse que vous avez constitué dans le groupe TD *A* le binôme *Yvette Durand* et *Georgette Dupond*. Votre production doit être placée dans un répertoire dont le nom est constitué du nom du groupe TD et des noms de famille du binôme par ordre alphabétique (donc **./A-Dupond-Durand/**).

Ce dossier comprend :

- un fichier "lisez-moi.txt" indiquant où se trouvent le rapport et les sources
- un rapport reprenant de manière synthétique vos choix, ce que vous avez réalisé, les tests, les performances et votre compréhension des optimisations que vous avez mise en œuvre
- les sources en C++ (identifiez clairement les fichiers que vous avez produits)
- les scripts de création
- les traces de validation du fonctionnement
- les mesures de performances
- des données significatives produites

Ce dossier est à archiver (sous le nom **A-Dupond-Durand.tar.bz2**) et à envoyer via l'outil de transfert de fichier de Renater `https://filesender.renater.fr/` à votre encadrant. Le courriel que vous enverrez aura **obligatoirement** comme objet : "**[M4105C] Rendu TP : binôme** *A-Dupond-Durand*" (en adaptant les identifiants à votre binôme ☺).

1.2 L'illustration

Ce document fait appel à des illustrations animées que vous retrouvez sur le site `http://m4105.merciol.fr/`ou dans le kit "m4105c-src.tar.bz2" à télécharger sur le même site.

1.3 La documentation

Ce module d'enseignement est principalement issu de travaux de recherches menés au sein de l'équipe Obelix de l'IRISA de Vannes [1] [2].

Pour aller plus loin dans le langage utilisé, vous pouvez emprunter à la Bibliothèque Universitaire des livres sur le langage C++ [3]. Vous pouvez également accéder au site de référence `http://www.cplusplus.com/reference/`.

A noter que pour une bonne pratique, il est nécessaire de connaître plus que le langage (qu'on réduit trop souvent à sa syntaxe). Il faut maîtriser les usages. Par exemple, un ensemble de fonctions est disponible en libre (comme *BOOST*) et répondent à de nombreuses situations : `http://www.boost.org/users/index.html`.

Enfin, il existe des réponses aux questions que tout le monde se pose. Via un moteur de recherche classique (framabee, duckduckgo, ...) on trouve des réponses pertinentes et appropriées ici : `https://stackoverflow.com/questions/tagged/c%2b%2b`.

Bonne route sur les chemins des algorithmiques efficaces...

L'utilisation du langage C++ dans ce module peu être vu comme une difficulté supplémentaire, vous l'avez peut pratiqué en DUT. Le premier TP sert de mise en jambe, comme une transition depuis le langage Java que vous pratiquez déjà.

2 TP1 : Micro-Parallélisme & gestion de cache

Les objectifs de ce TP sont de :

- découvrir l'environnement de développement du module
- expérimenter C++
- utiliser des outils d'évaluation de fonctions C++
- comprendre que les optimisations efficaces sont stratégiques (algorithmiques) et moins tactiques (style de programmation)
- découvrir des complexités de traitement inférieures à N et pourtant exhaustifs
- prendre conscience que d'autres optimisations matériel peuvent entrer en conflit avec vos choix

Si l'on aborde des algorithmes linéaires, c'est bien pour parler d'efficacité. On peut toujours réaliser des optimisations tactiques en adoptant un style de programmation épuré. Par exemple, on peut remplacer l'utilisation de `i++` par `++i` partout où c'est possible. En effet, dans le premier cas, une variable temporaire est conservée au cas où l'on souhaiterait utiliser le résultat de cette instruction. On peut également sortir un calcul dont la valeur est constante d'une boucle. Mais la plupart du temps, ces optimisations peuvent être automatisées et ont déjà fait l'objet d'une intégration dans le compilateur (avec l'option `-O2`, voire `-O3`).

Les optimisations les plus efficaces sont stratégiques. Elle portent sur la complexité des traitements. Si l'on veut examiner tous les éléments à traiter, cette complexité sera au minimum en $O(n)$. Nous cherchons à faire en sorte que ce soit également notre maximum.

Cependant, il peut arriver que l'on puisse descendre en dessous de ce seuil, sans pour autant être dans la divination. C'est l'objet de ce TP.

2.1 Comptage des 1

Le défi de ce TP consiste à compter le nombre de bits dans un entier de 8 octets (ce qui correspond au type `uint64_t` en langage C) en un minimum d'opérations. Voici les résultats attendus :

valeur hexadécimale	**valeur binaire**	**résultat**
0x0... 0000	0x00000000 ... 00000000 00000000	0
0x0... 0001	0x00000000 ... 00000000 00000001	1
0x0... 0100	0x00000000 ... 00000001 00000000	1
0x0... 0042	0x00000000 ... 00000000 01000010	2
0x0... 0070	0x00000000 ... 00000000 11110000	4

Tab. 1: Somme des bits à 1

Question : Écrivez ou dessinez un algorithme qui cumule les bits (i.e. réalise ce traitement).

Réponse : ✍

Question : Quelle est la complexité de votre algorithme ? (Continuez à l'améliorer jusqu'à il soit en O(n).)

Réponse : ✍

Question : Écrivez maintenant cet algorithme en C++. On ne demande que le contenu de la fonction, c'est donc un code proche du langage Java.

Réponse : ✍

Nous allons passer à la phase de réalisation, qui comprendra une étape de test et une étape d'évaluation de performance. Pour ce faire nous allons découvrir quelques outils qui vous sont offerts.

De nombreux fichiers sont préfixés du nom *merciol*. Il ne s'agit pas de mégalomanie. Cela permettra de

```
m4105c-A-Dupond-Durand/
├── build
│   └── lib
│       └── libmerciol.a
├── src
│   ├── include
│   │   ├── misc
│   │   │   ├── merciolChrono.hpp
│   │   │   ├── merciolTest.hpp
│   │   │   ├── merciolChrono.tpp
│   │   │   └── merciolTest.tpp
│   │   └── tp1
│   │       └── merciolAnswer.hpp
│   └── cpp
```

bien différencier, dans l'archive que vous allez rendre, votre production. Car, vos fichiers y seront facilement identifiables.

Vous devez télécharger le kit de démarrage "*m4105c-src.tar.bz2*" du module M4105C sur `http://m4105.merciol.fr/`.

Il comprend une arborescence de fichiers (voir la figure 1) dont voici les principaux éléments :

- `build` est le répertoire de production des exécutables. Il contient en particulier :
  - `lib/libmerciol.a` qui inclut une solution déjà compilée
  - `out/microPar` le futur programme exécutable que vous devez produire
- `src` est le répertoire des fichiers sources. Il se sépare en deux :
  - `include` les fichiers entêtes qui ne produisent pas d'exécutable.
    - `merciolChrono.hpp` déclare les fonctions qui permettent de réaliser des mesures de temps d'exécution. Lisez attentivement les commentaires du fichiers.
    - `merciolTest.hpp` déclare les fonctions qui permettent de réaliser des tests de validité et de performances. Lisez attentivement les commentaires du fichiers.
    - `merciolChrono.tpp` et `merciolTest.tpp` sont les définitions en ligne des fonctions précédentes. Le choix d'écriture en ligne a deux raisons :
      - perturber a minima les mesures de performances
      - permettre l'écriture de n'importe quelle fonction à tester, tous les codes génériques devant être connus lors de la phase de compilation avant l'édition de lien.
    - `merciolAnswer.hpp` est l'entête des solutions demandées. Leurs sources ne sont bien évidemment pas fournies. Mais, elles ont été compilées dans la bibliothèque "libmerciol.a".
  - `cpp` les sources qui produiront des exécutables.
    - `microPar.cpp` est le squelette du programme que vous devez réaliser durant ce TP.
- `Makefile` qui simplifiera les étapes de production C++. Il contient principalement les entrées suivantes (voir la commande `make -n -p`) :
  - `make clean` supprime les fichiers temporaires
  - `make wipe` supprime en plus les fichiers produits (ne garde que les fichiers sources)
  - `make microPar` crée un exécutable

Mise au point d'un programme en langage C++

Il existe un outil de mise au point des programmes écrits en C ou C++ : *gdb*. Il faut au préalable avoir compilé le vôtre avec l'option "`-g`".

Au moment de l'exécution, faîtes précéder la commande shell de "`gdb -arg `". Attendre le prompt gdb et saisissez "`run`".

La commande "`where`" permet de savoir où l'on se trouve dans la pile d'exécution.

La commande "`frame n`" permet de se placer dans la pile d'exécution.

Il est possible de placer des points d'arrêt et d'afficher des valeurs des variables avec la commande "`print`".

Question : Modifiez le fichier `microPar.cpp` en introduisant votre solution dans le corps de la fonction `ones64seq`. Compilez et exécutez jusqu'à qu'il n'y ait plus d'erreur (message "`0 error`"). Recopiez le code source de votre programme et la trace obtenue.

Réponse : ✍

L'affichage doit correspondre à ceci :

```
0 error

Time         Sum              Count Mean             Min              Max
       test 00:00:00.182039 1000  00:00:00.000182 00:00:00.000153 00:00:00.001817
```

List. 1: Temps d'exécution

Puisque le temps nécessaire à l'exécution de 1 000 microTests est en moyenne de 168 microsecondes, cela signifie que le temps moyen d'exécution d'un appel est de 168 nanosecondes.

Question : Ajoutez dans le fichier source `microPar.cpp` la ligne

```
1  perfFunction ("ref", ones64, 1000, 1000);
```

List. 2: Test référence

Compilez, exécutez, regardez, recopiez le résultat ci-dessous, comparez et... allez pleurer.

Réponse : ✍

Comment diviser votre temps d'exécution par 20 ? C'est maintenant que commence le travail proprement dit de ce TP.

Vous avez a priori regardé chaque élément une seule fois. On ne peut pas faire moins, sinon ce serait de la divination.

Mais, nous allons chercher à réaliser plusieurs traitements en une seule instruction. Regardons comment fonctionne la somme de 2 bits.

valeur	**somme décimale**	**somme binaire**
00	0	00
01	1	01
10	1	01
11	2	10
ab	a + b	xx

Tab. 2: Somme de 2 bits

Nous aurions pu également dire que nous posions l'opération des bits pairs (a) au dessus des bits impairs (b) :

```
ab & 10 >> 1   a0 >> 1   0a
ab & 01 >> 0   0b >> 0   0b
                         --
                         xx
```

Tab. 3: Somme séparée des bits pairs et impairs

Cela peut même se faire sur plusieurs segments d'une série de bits simultanément.

```
abcdefgh & 10101010 >> 1   a0c0e0g0 >> 1   0a0c0e0g
abcdefgh & 01010101 >> 0   0b0d0f0h >> 0   0b0d0f0h
                                           --------
                                           xxyyzztt
```

Tab. 4: 4 sommes de 8 entiers de 1 bit

xx, yy, zz et tt sont respectivement les sommes des bits ab, cd, ef et gh. Leurs valeurs étant inférieures à 2, ils ne prennent pas plus de place que les couples d'origines.

Nous pouvons faire la somme de xx avec yy et de zz avec tt suivant le même principe. La somme de 2 nombres inférieurs ou égale à 2 est inférieur ou égale à 4. Il faut donc 3 bits seulement pour chaque sous-résultat.

```
xxyyzztt & 11001100 >> 2   xx00zz00 >> 1   00xx00zz
xxyyzztt & 00110011 >> 0   00yy00tt >> 0   00yy00zz
                                           --------
                                           0iii0jjj
```

Tab. 5: 2 sommes de 4 entiers de 2 bits

Une troisième étape permet de faire la somme des résultats précédents. La somme de 2 nombres inférieurs ou égal à 4 est inférieur ou égale à 8. Il faut donc au maximum 4 bits pour ce résultat.

```
0iii0jjj & 11110000 >> 4   0iii0000 >> 1   00000iii
0iii0jjj & 00001111 >> 0   00000jjj >> 0   00000jjj
                                           --------
                                           0000kkkk
```

Tab. 6: 1 somme de 2 entiers de 4 bits

Dans ce cas, n étapes permettent de faire la somme de 2^n bits. On peut donc créer un algorithme qui réalise la somme de 64 bits en $log_2(64)$ étapes. La complexité est de $O(log_2(n))$ et pour autant nous avons bien considéré les n bits de l'entier sans aucune omission.

Question : Faîtes un nouvel algorithme qui réalise simultanément sur un mot de 64 bits :

- 32 sommes de 1 bit
- 32 sommes de 2 bits
- 16 sommes de 4 bits
- 8 sommes de 8 bits
- 4 sommes de 16 bits
- 2 sommes de 32 bits

Pensez à écrire le masque en hexadécimal, c'est plus lisible.

Réponse : ✍

Question : Vérifiez les performances et recopiez les traces.

Réponse : ✍

2.2 Position du “1” le plus fort

Le défi cette fois est de déterminer la position du bit le plus élevé (donc à gauche) dans un entier de 8 octets (ce qui correspond au type `uint64_t` en langage C) et ce, en un minimum d'opérations. Voici les résultats attendus :

valeur hexadécimale	**valeur binaire**	**résultat**
0x0... 0000	0x00000000 ... 00000000 00000000	0
0x0... 0001	0x00000000 ... 00000000 00000001	1
0x0... 0100	0x00000000 ... 00000001 00000000	9
0x0... 0042	0x00000000 ... 00000000 01000010	7
0x0... 0070	0x00000000 ... 00000000 11110000	8

Tab. 7: Position du bit le plus fort

Question : Écrivez ou dessinez un algorithme qui indique la position du bit de poids fort (i.e. réalise ce traitement).

Réponse : ✍

Question : Écrivez maintenant cet algorithme en C++. On ne demande que le contenu de la fonction, c'est donc un code proche du langage Java.

Réponse : ✍

Question : Modifiez le fichier `microPar` en introduisant votre solution dans le corps de la fonction `size64seq`. Compilez et exécutez jusqu'à qu'il n'y ait plus d'erreur : “`0 error`”. Notez le programme définitif et la trace.

Réponse : ✍

Question : Ajoutez dans le fichier source `microPar.cpp` la ligne

```
1  perfFunction ("ref", size64, 1000, 1000);
```

List. 3: Test référence

Compilez, exécutez, regardez, recopiez le résultat ci-dessous, comparez et... décidément c'est une manie. Arrêtez de pleurer !

Réponse : ✍

Cette fois la solution consiste à remplir tous les bits à droite de celui le plus fort. Regardez ce qui se passe avec une valeur aléatoire x avec les commandes successives suivantes :

- "$x =>> 1$"
- "$x =>> 2$"
- "$x =>> 4$"

Question : Observez et expliquez ce phénomène.

Réponse : ✍

Question : Faîtes un nouvel algorithme qui :

- complète les bits de poids faible à 1
- compte le nombre de bits à 1 avec la solution précédente

Pensez à écrire le masque en hexadécimal, c'est plus lisible.

Réponse : ✍

Question : Vérifiez les performances et recopiez les traces.

Réponse : ✍

2.3 Gestion de cache

Le but de cette section est de réaliser qu'il y a des limites à l'optimisation. Nous venons de le voir certains traitements peuvent être réalisés en parallèle. Les architectures des processeurs aujourd'hui sont très poussés. Les

compilateurs exploitent ces propriétés. Comme vous l'avez appris en module M3101, le processeur peut utiliser des mémoires caches bien plus rapides que la mémoire vive principale. La différence d'accès est telle qu'il peut y avoir un facteur non négligeable entre réaliser une boucle sur la mémoire cache et remplacer cette mémoire. Nous allons l'illustrer avec un exemple tiré de http://igoro.com/archive/gallery-of-processor-cache-effects/.

Supposons que nous réalisions une boucle sur un tableau. Il faut que le traitement soit réel (additionner zéro risque d'être interprété comme ne rien faire par le compilateur et le temps de traitement sera nulle). Il faut également que la boucle soit assez grande pour qu'une modification soit visible (ici 64 Mo). Le listing 4 donne la déclaration globale du tableau.

```
1 static const size_t dim = 64 * 1024 * 1024;
static int tab[dim];
static int k = 1;
```

List. 4: Déclaration pour test de mémoire cache

Nous allons simplement ajouter 3 à chaque élément et imaginer que nous avons imaginé une optimisation qui nous permet de ne traiter qu'un élément sur deux ou plus. Le listing 5 donne l'exemple d'une boucle traite tous les éléments.

```
1 // Boucle pas 1
for (size_t i = 0; i < dim; i += 1)
  tab[i] *= 3;
```

List. 5: Boucle à pas de 1

Le listing 6 donne l'exemple d'une boucle qui ne traite qu'un éléments sur 16.

```
1 // Boucle pas 16
for (size_t i = 0; i < dim; i+ = 16)
  tab[i] *= 3;
```

List. 6: Boucle à pas de 16

Les résultats indiqués vont dépendre du processeur employé (les caractéristiques sont fournies par la commande "`lscpu`"). En théorie la 2de boucle devrait être 16 fois plus rapide. Avec le processeurs décrit figure 2, le temps de la 2de boucle n'est que 1,10 fois plus rapide (seulement 10 % de gain). Comment est-ce possible ?

Modèle :	Intel i5-7440HQ
Vitesse :	799.975 MHz
Vitesse max :	3800,0000 MHz
Vitesse min :	800,0000 MHz
Cache L1d :	32 Ko
Cache L1i :	32 Ko
Cache L2 :	256 Ko
Cache L3 :	6144 Ko

Fig. 2: Caractéristiques du processeur

Il faut en chercher la raison dans la gestion du cache mémoire. L'accès à la mémoire principale est "tellement" lent qu'on aurait le temps de faire 16 itérations dans la mémoire cache avant d'accéder aux blocs de données suivants dans la mémoire principale. Dans ces conditions, une fois qu'un bloc est présent, il ne faut pas se gênez pour l'utiliser. C'est le genre de situation où une optimisation ne sert de rien. Donc pour ne pas vous arracher le cheveux pour rien, évaluez bien les performances de chaque parties de vos algorithmes pour savoir celles qui méritent d'être étudiées.

La figure 3 fournit les temps d'exécution pour les pas de 1 (2^0) à 1024 (2^{10}). On constate un temps de calcule proche entre 1 et 16.

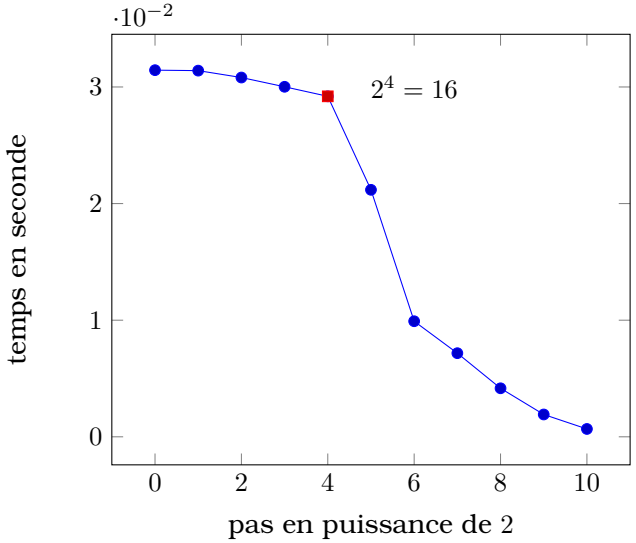


Fig. 3: Temps d'exécution suivant le pas

Question : Réalisez une mesure de temps pour un pas de 1 et un pas de 16.

Réponse : ✍

Question : Quel est le gain en pourcentage de temps de calcul ?

Réponse : ✍

Question : Désignez la courbe de 1 à 1024. Indiquez les caractéristiques de votre processeur et concluez.

Réponse : ✍

2.4 Identifier une 2^d niveau de cache

La section précédente nous a montré que l'on peut identifier un premier niveau de cache mémoire. Il a suffit d'accéder à toute les cases d'un tableau et de se rendre compte qu'un fois un bloc de plusieurs case en mémoire cache, leur accès été rapide.

Nous allons voir qu'il existe un 2^d niveau de cache. Pour cela nous allons faire varier la taille d'un tableau et tenté de parcourir plusieurs fois l'ensemble des valeurs. Pour aller plus vite nous bondirons de 16 cases à chaque fois. Au bout du tableau, on reviendra au début. Dans cet exercice tant que le tableau et complètement contenu dans le cache de premier niveau, l'accès sera constant. Au delà se sera plus lent. Mais tant que l'on restera dans une taille disponible dans le second niveau, l'accès restera raisonnable.

Le listing 8 permet de disposer d'un tableau assez grand. Nous n'utiliserons que le début à chaque fois en faisant varier le nombre de cases réellement utilisées. La variable "mod" est une astuce pour ne pas influence le temps de calcul. Comme nous voulons retourné au début du sous tableau en fonction de sa taille, nous devrons réaliser un modulo. Or si notre tableau à une taille correspondant à une puissance de 2, il suffit de faire un masque pour supprimer les bits de poids fort.

```
1 static int tab[64 * 1024 * 1024];
  static int mod = 1024-1;
  static int steps = 64 * 1024 * 1024;
```

List. 7: Déclaration pour test 2^d niveau

Le listing **??** présente le cœur du test. Une boucle on l'on accède à tous les éléments puis on recommence.

```
1 loop () {
    for (int i = 0; i < steps; ++i)
      ++tab [(i * 16) & mod]; // (x & mod) is equal to (x % mod+1)
  }
```

List. 8: Déclaration pour test 2^d niveau

La figure 4 fournit les temps d'exécution pour des tableaux dont la taille varie entre $1ko$ (2^{10}) à $64Mo$ (2^{26}). On constate cet fois 2 paliers.

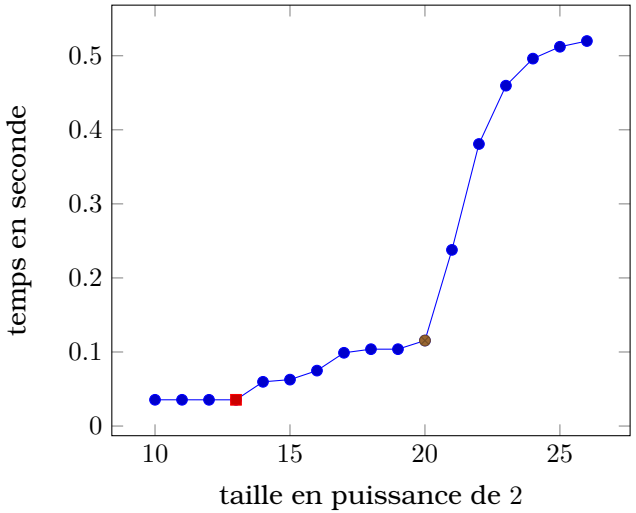


Fig. 4: Temps d'exécution suivant la taille

Question : Désignez la courbe de 2^{10} à 226 (en faisant simplement varier la puissance de 10 à 26).

Réponse : ✍

Question : Indiquez les caractéristiques de votre processeur. Essayez d'interpréter les résultats.

Réponse : ✍

Page de notes personnelles : ✍

3 Traitement d'images

A partir de maintenant dans ce module, tous les exercices porteront sur le traitement d'images. Nous allons construire des structures de données hiérarchiques (sous forme d'arbres informatiques) à partir d'images en niveaux de gris.

3.1 Présentation des arbres

Pour vous convaincre de l'intérêt de telles pratiques, voici une présentation de traitements réalisés sur des images satellites. Les algorithmes associés à ces traitements sont issus de la recherche en informatique.

Nous travaillerons sur une image satellite de Nairobi (voir figure 5) de 8 000x8 000 pixels (soit 64 millions de pixels). L'image d'origine est sur 16 bits, mais celle qui vous est fournie a été convertie sur une profondeur de 8 bits (1 octet) pour simplifier les traitements. Malgré ce régime, l'image demeure tout de même d'un poids de 64 Mo. Cette image est accessible sur l'espace forum `/ubs/forum/prof/2tin01/M4105C/mairobi-byte.tif`. Ne la copiez pas dans votre répertoires, mais créez un lien dessus.

Fig. 5: image satellite de Nairobi

```
1  ln -s /ubs/forum/prof/2tin01/M4105C/mairobi-byte.tif .
```

List. 9: Message de la classe `Option`

Pour vous donner une bonne idée de ce qu'est une représentation sous forme d'arbres d'une image, nous allons considérer une métaphore qui s'appuie sur le relief d'un territoire. Vous connaissez les carte IGN sur lesquelles figure les courbes de niveau (d'altitude en fait). Imaginez qu'il y ait une inondation (genre déluge) avec une eau colorée. Le niveau atteint par la montée des eaux laisserait une trace sur les murs des maisons, sur tout le paysage. Au début vous allez voir apparaître une dizaine de lacs. Puis, au fur et à mesure que l'eau monte, les lacs s'étendent et se rejoignent. A la fin, il n'y a plus qu'une seule surface.

L'arbre est la structure qui représente ces étapes. Les lacs sont les feuilles. La surface la plus haute est la racine. Dans les étapes intermédiaires des nœuds de l'arbre connectent des lacs du niveau inférieur ainsi que de nouvelles feuilles qui apparaissent.

Dans notre cas, on ne considère pas l'altitude en un point, mais la valeur du pixel, son niveau d'intensité (noir = faible intensité, blanc = forte intensité).

Fig. 6: Types d'arbre

Lorsque les feuilles sont noires et la racine est blanche, on appelle ces arbres des max-trees (car à chaque étape on considère la valeur maximum de la zone). Lorsque les feuilles sont blanches et la racine noire, on appelle ces arbres des min-tree (car à l'inverse on considère la valeur minimum des zones). Il est également possible de prendre comme racine la valeur médiane (les feuilles étant noires ou blanches en s'écartant de la racine). Ces derniers arbres sont nommés arbres de formes (tos-tree). La figure 6 compare ces différents arbres. Jusqu'à la fin de ce document, nous ne traiterons que des max-trees.

3.2 Utilisation des arbres

La figure 7 n'est qu'une illustration. En tout rigueur, l'inclusion est fausse car le traitement sur une image couleur est bien plus complexe (du fait du choix de relation d'ordre entre pixels). Mais, l'illustration permet de comprendre l'intérêt de tels arbres. En ayant hiérarchisé les objets de ce bureau vu de dessus par niveau de luminosité, on obtient du plus sombre au plus clair :

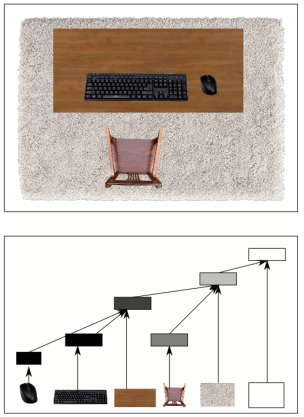


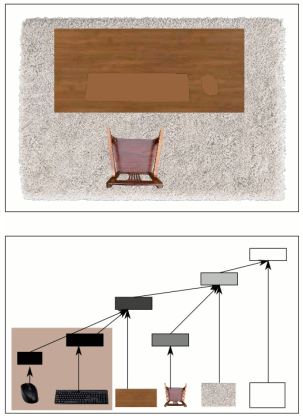
Fig. 7: Max-tree

- souris
- clavier
- chaise
- table
- tapis
- sol

A supposer que l'on puisse classer facilement les pixels comme appartenant à une de ces catégories, nous obtiendrions l'arbre sous la photo dans la même figure.

En filant la métaphore précédente des courbes de niveau, nous pourrions inonder les parties les plus basses (sombres) en étêtant cet arbre (i.e. en élaguant certaines branches). C'est ce que représente la figure 8. Deux remarques,

- la première est que vous comprenez que cet exemple est faux et n'est qu'une illustration. Nous ne pouvons pas reconstituer les objets masqués par ceux que l'on supprime en avant-plan.
- La seconde est que, avec ce procédé, nous ne pouvons que faire disparaître les objets les plus sombres (ou clair avec un min-tree) mais pas les objets "gris".



C'est pourtant ce procédé que nous allons étendre et généraliser. Nous allons ajouter une information supplémentaire à chaque nœud. Par exemple, un attribut de taille en pixel de la zone couverte par le nœud. Ensuite, nous pourrons faire disparaître les objets inférieurs à une taille donnée comme nous pouvions faire disparaître les objets sombres.

Nous pouvons même aller plus loin. En réalisant la différence entre une image dont on a supprimé les objets inférieurs à X et une autre avec la valeur Y. Cela permet de faire surgir tous les objets de tailles comprises entre X et Y.

Fig. 8: Max-tree étêté

Pour illustrer ce qui vient d'être vu, l'animation `M4105C-1-ap` se trouve dans le répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-1-ap.gif`. Voici le détail des diapositives constituant l'animation :

- 1 : création des attributs de taille par nœud
- 2 : suppression des nœuds inférieurs à 50
- 3 : suppression des nœuds inférieurs à 400
- 4 : différence entre l'image complète et celle supprimant les nœuds inférieurs à 50
- 5 : différence entre l'image supprimant les nœuds inférieurs à 50 et celle supprimant les nœuds inférieurs à 400

3.3 Principe de construction d'arbre

Pourquoi partir sur une telle technologie ? Parce que nous avons réussi à construire de tels arbres très rapidement. L'identification "d'objets" dans une image devient donc efficace.

Connectivité

La construction d'un arbre s'effectue en considérant le voisinage proche de chaque pixel. On appelle cela la connectivité. On dira que la connectivité est de 4 si l'on considère uniquement les 4 voisins d'un pixel, suivant les parallèles au côté de l'image. On dira que la connectivité est de 8 si en plus on ajoute les voisins des directions obliques. En réalité, on ne traite jamais que la moitié des voisins, car les relations min, max, ... sont symétriques.

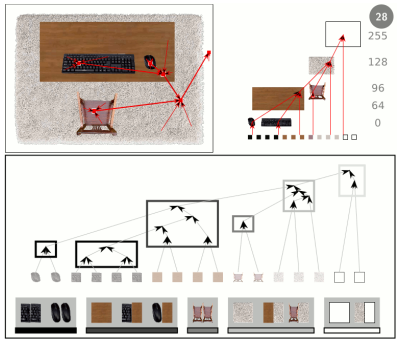


Fig. 9: M4105C-2-build-tree

Une animation est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-2-build-tree.gif`.

La figure 9 se divise en 3 régions :

- en haut à gauche, l'image à analyser. On y a représenté des "pixels maîtres" qui, pour nous, matérialiserons les nœuds de l'arbre produit.
- en haut à droite, une représentation simplifiée de l'arbre produit.
- en bas, les étapes de construction.

Nombre maximum de nœuds

Lors de la montée des eaux, un nœud n'est créé que si la zone s'étend sur de nouveaux pixels. La création d'un nœud "consomme" au moins un pixel. Le nombre de nœuds est nécessairement strictement inférieur au nombre de pixels pour les arbres que nous manipulons.

L'animation est constituée de 28 diapositives reprenant l'enchaînement des opérations de construction. Cet algorithme s'inspire de "Union-Find" imaginé en 1975 par R.Tarjan [4].

- 1 : identification des pixels.
- 2 : considération des voisins gauches et bas, pour une connectivité 4.
- 3 : choix de la fonction de pondération (ici maximum de 2 voisins).
- 4 : constitution de la liste des paires de voisins avec leurs poids triés de façon croissante.

- 5-9 : connexion des feuilles et nœuds du niveau le plus bas
- 10 : fusion des nœuds “frères” temporaires appartenant à une même zone.
- 11-16 : connexion du niveau suivant
- 17-18 : connexion du niveau suivant
- 19-23 : connexion du niveau suivant
- 24-26 : connexion du niveau suivant

L'algorithme est presque linéaire, mais il reste deux difficultés :

- la constitution de la liste triée des voisins (les algorithmes de tri largement utilisés sont en $O(n)$).
- la fusion des nœuds “frères” temporaires appartenant à une même zone

Ces deux points seront abordés par la suite. A la fin de ce module vous aurez réécrit la construction d'un arbre de manière efficace.

Les 6 TP suivants sont autant d'étapes pour réaliser le programme d'analyse d'images à rendre. Dans chacun d'eux, vous aurez à surcharger les méthodes d'une classe que vous ferez dériver de la classe `ArrayTree` dans le répertoire `tree`.

Chaque fois, le TP abordera un aspect particulier intervenant dans l'analyse. Les TP ne sont pas de longueur égale. Il ne faut donc pas imaginer faire un TP et un seul par semaine. Ce sont des guides. Sachez prendre de l'avance quand c'est possible. Nous commençons par nous familiariser à la lecture d'images en C++ et leurs représentations sous forme de tableaux d'entiers.

4 TP2 : Lecture d'images

Les objectifs de ce TP sont de :

- lire des images avec la librairie GDAL
- utiliser de l'API offerte
- personnaliser de l'environnement de développement
- user d'un temps d'analyse et de planification du travail sur l'ensemble du module

Ce TP n'est pas complexe et ne fait pas intervenir de concepts nouveaux. Il faut en profiter pour prendre de l'avance et préparer la structure de ce qui sera à rendre en fin de module.

Plusieurs outils sont offerts. Nous allons dans un premier temps les découvrir. Puis, la fin du TP consistera à les utiliser.

4.1 GDALImage et Raster

Nous allons utiliser la librairie GDAL (`http://www.gdal.org/`) dont vous trouverez une documentation en ligne pour C++ sur `http://www.gdal.org/classGDALDataset.html`.

Il faut comprendre qu'une image peut être constituée de :

- 1 bande pour une image en niveau de gris
- 3 bandes pour une image en couleur
- plusieurs bandes pour une image multi-spectrale
- plusieurs centaines de bandes pour une image hyper-spectrale (le cas de certaines images satellites)

Les bandes représentent une matrice (donc un tableau à 2 dimensions) d'entiers de taille identique (de 1 octet à 8 octets selon la précision du capteur), de valeurs réelles voire complexes (pour des usages très particuliers, comme dans le cas d'images satellites polarisées). GDAL va nous permettre de lire les bandes indépendamment les unes des autres. Nous ne considérons dans ce module que les images d'une seule bande comportant des entiers sur 1 octet.

Voici les principales fonctionnalités de GDAL :

- lecture d'un en-tête d'image. Qui donne accès aux dimensions de l'image, au nombre de bandes et à la nature des pixels.
- lecture d'une bande. Qui remplit une matrice d'entiers fournie en paramètre.
- écriture d'un en-tête d'image. Qui fixe les dimensions, le nombre de bandes et la nature des pixels, ainsi que la création du fichier qui contiendra l'image.
- écriture d'une bande. Qui transforme la matrice d'entier fournie en paramètre dans le format de l'image créée.
- réservation de ressources pour la librairie. Qui doit être réalisée avant tout appel à la librairie GDAL.

- libération des ressources. Qui doit être réalisée après le dernier appel à la librairie GDAL.

La classe `GDALImage` est offerte et se trouve dans le répertoire `misc`. Dès la première création d'un exemplaire de cette classe, les ressources GDAL sont réservées automatiquement. Elles seront libérées à la destruction du dernier exemplaire de cette classe.

La majorité des paramètres sont masqués et cette classe ne traitera que d'images en niveau de gris, de 8 bits de profondeur (dont les pixels sont des valeurs entières sur 1 octet de 0 à 255).

Pour simplifier l'allocation mémoire de matrices de pixels, la classe `Raster` est fournie. Elle alloue la mémoire pour une matrice de pixels suivant les dimensions données en argument du constructeur. La mémoire sera libérée à l'appel du destructeur.

Voici un exemple d'utilisation de la classe `GDALImage` :

```
1  GDALImage image (fileName);
   Size size (image.getSize ());
   Raster raster (size);
   image.readBand (0, raster.getPixels (), NullPoint, optionsize);
```

List. 10: Utilisation de `GDALImage`

La première ligne crée un exemplaire et donc initialise l'utilisation de la librairie GDAL. Le nom de fichier en paramètre lit l'en-tête de l'image (dimension, nombre de bandes, ...).

La troisième ligne réserve une matrice d'entiers qui recevra les valeurs des pixels.

La quatrième remplit la matrice à partir de la bande numéro 0. On fournit à cette lecture le coin haut gauche d'un rectangle de découpe et sa taille. En donnant le point nul (coordonnées $(0, 0)$) et la taille de l'image, c'est toute l'image qui est lue.

4.2 Syntaxe de *lambda* fonction

Dans le code qui vous est fourni, certaines déclarations prennent des fonctions en paramètre. Vous avez déjà pratiqué cette technique en Java, en créant par exemple des tâches (*Threads*) anonymes, dans lequelles vous spécialisiez la méthode *run* à la "volée".

En C++ on peut passer en argument une fonction anonyme dite fonction *lambda*. La syntaxe est simple et de la forme :

```
1  [liste des variables locales transmises] (liste des parametres de la fonction) {
     corps de la fonction
   }
```

List. 11: Syntaxe de fonction *lambda* C++

Voici un exemple d'utilisation :

```
1  template<typename Funct>
   void f1 (const Funct &lambda) {
     for (int i = 0; i < 10; ++i)
     lambda (i);
5  }

   void
   MaClasse::appelant () {
     int variableLocale;
10   f1 ([this, variableLocale] (int x) {
       this->attr = variableLocale+x;
     });
   }
```

List. 12: Exemple d'utilisation de fonction *lambda* C++

La liste des variables locales provient de l'appelant. Dans l'exemple précédent, il s'agit de *this* et *variableLocale*.

La liste de paramètres sera fournie par f1. Dans l'exemple précédent, il s'agit de *x*.

4.3 GraphWalker

Lorsque l'on crée un arbre à partir d'une matrice de pixels, il peut être pratique de disposer de fonctions de parcours de tous les pixels ou de tous les voisins des pixels.

Dans notre cas, les pixels forment les sommets de notre graphe. On les appelle *vertex*. Les voisins forment les arcs reliant 2 sommets. On les appelle *edges*.

Les pixels peuvent être adressés de deux manières :

- par leurs coordonnées cartésiennes (x, y)
- par leur index commençant à 0 pour les coordonnées $(0, 0)$, puis 1 pour $(1, 0)$ jusqu'à $n - 1$ pour $(width - 1, height - 1)$. L'incrément se fait prioritairement sur l'axe des x. Une fois la largeur atteinte, on reprend $x \leftarrow O$ en incrémentant y.

La classe `GraphWalker` se trouve dans le répertoire `tree`.

Voici le rôle des méthodes de cette classe :

- `vertexMaxCount` : le nombre de pixels
- `edgeMaxCount` : le nombre de voisins
- `forEachVertexIdx` : applique sur tous les index, une fonction *lambda* prenant un index de sommet.
- `forEachVertexPt` : applique sur toutes les coordonnées, une fonction *lambda* prenant des coordonnées de sommet.
- `forEachEdgePt` : applique sur tous les voisins en connectivité 4, une fonction *lambda* prenant les coordonnées de deux voisins.

Voici un exemple d'utilisation d'un exemplaire de `GraphWalker` pour faire des mesures statistiques sur tous les pixels d'une image.

```
graphWalker.forEachVertexIdx ([&pixels, &pixelStat] (const DimImg &leafId) {
  pixelStat (pixels [leafId]);
});
```

List. 13: Utilisation de GraphWalker

4.4 Option

Voici enfin une dernière classe utile pour ce TP : la classe `Option`. Elle prend en charge toute l'analyse des paramètres fournis à l'appel de votre programme lancé via l'interpréteur de commande.

Nous partons du principe qu'il y aura toujours un fichier d'image en entrée et, éventuellement, un rectangle d'extraction *crop*, un fichier d'image en sortie et des entiers de paramétrage.

Si les paramètres fournis au lancement de votre programme ne correspondaient pas, il y aurait le message d'erreur suivant :

```
Usage:
build/out/monProgramme [options] inputFileName [outputFileName]

Allowed options:
--debug               debug mode
-b [ --band ] arg     select input band (first band is 0) (default 0)
-x [ --left ] arg     left crop (default center)
-y [ --top ] arg      top crop (default middle)
-w [ --width ] arg    width crop (default input width)
-h [ --height ] arg   height crop (default input height)
-p [ --param ] arg    customized arguments
--input-file arg      inputFileName [outputFileName]
```

List. 14: Message de la classe `Option`

Pour bénéficier de cette analyse automatique, il suffit de mettre dans le `main` de votre programme la déclaration d'un exemplaire de la classe `Option` :

```
#include "misc/merciolOption.hpp"

using namespace std;
using namespace merciol;

int
main (int argc, char **argv, char **envp) {
  Option option (argc, argv);

  // ... utilisation des options dans votre programme
  return 0;
}
```

List. 15: Utilisation de GraphWalker

Le code de la classe `Option` s'appuie, lui aussi, sur la librairie *Boost*.

4.5 Entraînement

Vous avez toutes les cartes en main. C'est à vous de jouer !

Question : Compilez et exécutez dans le répertoire `ws` le programme `imageStat` qui se trouve dans le répertoire `tp2` avec les arguments suivants : `../build/out/imageStat ../data/nairobi-byte.tif -x 100 -y 100 -w 100 -h 100`

Réponse : ✍

Vous devriez obtenir un résultat du genre :

```
Input data/nairobi-byte.tif: [8000,8000] (1 chanels of Byte)
Crop topLeft:(100,100) size:[100,100] band:0
Output :
Params:

Sum       Count     Mean      Min       Max
363940    10000     36.394    10        82
```

Question : Lisez et comprenez comment les fonctions statistiques sont appliquées à tous les pixels de votre sélection. Remarquez la concision des fonctions statistiques de *Boost*.

Réponse : ✍

Question : Compilez et exécutez dans le répertoire `ws` le programme imageFilter qui se trouve dans le répertoire *tp2* avec les arguments suivants : `../build/out/imageFilter -x 100 -y 100 -w 100 -h 100 -p 5 -p 0 ../data/10m.tif ../data/crop.tif` Vous obtiendrez une image réduite comme celle de la figure 10

Fig. 10: Sélection

Réponse : ✍

Question : Lisez et comprenez comment les fonctions de manipulation de pixels sont appliquées aux pixels de votre sélection.

Réponse : ✍

Question : Amusez-vous à mettre en négatif une partie de l'image ou à augmenter le contraste (i.e. faire que la valeur minimum soit à 0 et la valeur maximum à 255).

Réponse : ✍

Question : Que faut il faire pour augmenter l'intensité de la photo ? Ou pour la diminuer ?

Réponse : ✍

Félicitations, vous venez de créer votre premier programme de traitement d'images !

Page de notes personnelles : ✍

5 Représentation arborescente via des tableaux

Une structure arborescente peut-être représentée de diverses manières. Vous en connaissez déjà de nombreuses :

- structure XML (utilisée pour représenter le DOM de HTML)
- structure indentée (utilisée pour l'écriture de programme en python)
- à partir de listes
- à partir de vecteurs
- à partir de cellules pointant sur d'autres cellules ou des feuilles

Voici une nouvelle forme utilisant exclusivement des tableaux et le nombre de nœuds. Vous pouvez accéder à la version en couleur de la figure 11 dans votre répertoire `doc/images` ou pointée `http://m4105.merciol.fr/_media/supports/m4105c-array-tree.png`.

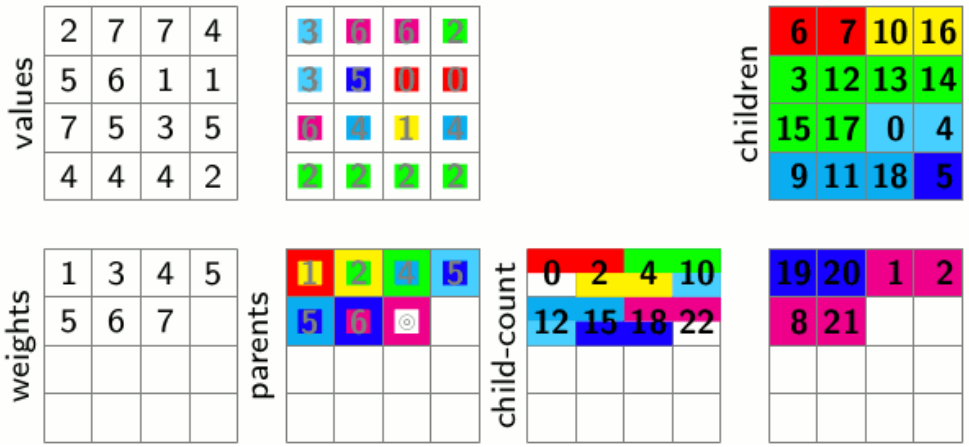


Fig. 11: M4105C-array-tree

L'image à analyser est représentée par le tableau en haut à gauche : *values*. Chaque case correspond à un pixel. Ce sont les feuilles de l'arbre.

Une case du tableau peut-être désignée par des coordonnées cartésiennes : la valeur à la case $(2, 1)$ vaut 1. Ou par un index : la case 6 vaut 1.

Comme les pixels ont une valeur, les nœuds ont un poids enregistré dans le tableau en bas à gauche : *weights*. Les parents sont désignés uniquement par un index.

Les deux tableaux juste à leur droite gèrent les pointeurs vers les parents. Le chiffre et la couleur au centre de la case désignent l'index du parent. Pour un parent, son index correspond à sa couleur extérieure.

La figure 12 montre la correspondance entre index d'un parent et sa couleur.

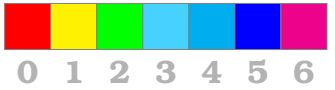


Fig. 12: Couleur des nœuds

L'ensemble des liens des parents des feuilles et des nœuds peut être vu comme une suite continue d'un même tableau à une dimension. La figure 13 représente les mêmes informations sur une seule ligne, avec en dessous l'index global correspondant.

Fig. 13: Représentation linéaire des liens feuilles vers racine

Le tableau 8 fournit les valeurs associées dans le cas de 7 parents :

Identifiant de la racine		6
Nombre de feuilles	largeur*hauteur	16
Nombre de nœuds	racine+1	7
Nombre d'éléments	nœuds+feuilles	23
Index du dernière élément	nombre d'éléments-1	22

Tab. 8: Correspondance des index

Le tableau suivant vers la droite *child-count* est reproduit de façon linéaire dans la figure 14. L'information contenue correspond aux index des liens vers les feuilles. Il a été constitué en comptant le nombre de fils pour chaque nœud. On retrouve l'index du début des fils d'un parent avec le même index que le nœud. L'index de fin est la case immédiatement à droite. La figure 14 indique les index des parents pairs dans la moitié supérieure et inférieure pour les parents impairs.

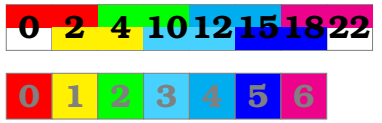


Fig. 14: Index des liens racine vers feuilles

Enfin, les deux derniers tableaux de droite représentent les liens vers les fils, donc de la racine vers les feuilles. De nouveau, ces tableaux sont rassemblés de façon linéaire dans la figure 15.

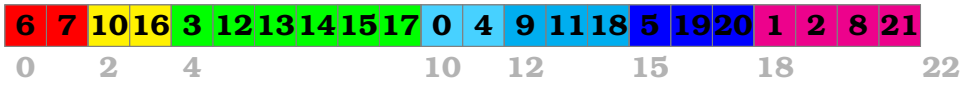


Fig. 15: Représentation linéaire des liens feuilles vers racine

Pour savoir si le fils est une feuille ou un nœud de l'arbre, il suffit de regarder si la valeur de l'index du fils est inférieur ou supérieur au nombre de pixels. Dans cet exemple, il suffit de comparer par rapport à la valeur 16.

Faisons quelques remarques sur cette organisation des données. Si l'on fait abstraction du recours à des mécanismes de compression (matrices à trou, ...), les tableaux utilisés ici ne prennent que l'exact besoin en mémoire. A titre d'exemple, si les fils d'un parent avaient été gérés par une liste ou un vecteur, nous aurions des informations supplémentaires de service (pointeur du premier élément, nombre d'éléments utiles dans l'ensemble, nombre de cellules restées vides avant une nouvelle allocation dynamique, ...). Dans notre situation, pas une seule information n'est superflu.

Autre remarque, il est possible (sans que cela soit imposé par la structure) de disposer d'une relation d'ordre dans les tableaux :

- Pour les parents :
  - de ne pointer que vers des parents d'index supérieur
  - et qu'ils soient triés par poids croissant
- Pour les fils :
  - que les pixels précèdent les nœuds
  - que les pixels soient triés par index (pixels en haut à gauche avant ceux en bas à droite)
  - que les nœuds soient triés par poids croissant
  - que les nœuds soient triés par index croissant

La technique de construction que nous allons réaliser dans ce module, possède les propriétés précédentes.

Ces propriétés sont indispensables pour permettre par la suite un parcours de l'arbre (nécessaire à la création d'attributs) qui soit linéaire et en parrallèle.

Par ailleurs, nous y reviendrons par la suite, toute la mémoire nécessaire peut être allouée dès le début du traitement. Contrairement à l'utilisation de vecteur qui, au fur et à mesure de l'ajout d'éléments, va allouer de nouveaux tableaux, recopier les valeurs précédentes et libérer les tableaux obsolètes.

Nous allons maintenant mettre en pratique cette structure.

6 TP3 : Structure minimale

Les objectifs de ce TP sont de :

- comprendre la structure en tableau
- mettre en place d'une classe dérivée dédiée à la structure
- réaliser d'un lanceur

Question : Commencez par faire votre classe dérivée qui n'ajoute aucune action.

Réponse : ✍

Question : Créez un lanceur. Vérifiez la compilation et l'exécution du programme.

Réponse : ✍

Question : Faîtes une initialisation "à la main" avec les valeurs en exemples dans ce document pour réaliser de futurs tests.

Réponse : ✍

Question : Prévoyez un dispositif de traces des éléments significatifs.

Réponse : ✍

Question : Réfléchissez à une autre structure et comparez le gain de mémoire obtenu avec celle proposée dans ce module.

Réponse : ✍

Page de notes personnelles : ✍

7 Structure triée

Nous avons indiqué que notre représentation sous forme de tableaux pouvait posséder des propriétés particulières. Dans cette section, nous abordons l'arrangement des parents par ordre croissant.

7.1 Tri par comptage

Dans les problèmes à surmonter nous avons mentionnés dans la section "3.3 Principe de construction d'arbre" l'obtention de l'ensemble des voisins triés par le poids de leur relation (max des valeurs des pixels).

Même si vous connaissez déjà la méthode de tri par comptage, décrivons-la à nouveau.

Dans la figure 16, nous prenons comme exemple des étoiles colorées à trier. Une animation correspondant à cet exemple est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-3-sort.gif`.

Fig. 16: M4105C-3-sort

Ces étoiles se différencient par leur couleur (de rouge à bleu) et leur nombre de branches (de 3 à 5). Comme nous le voyons au départ, les étoiles sont déjà triées par couleur. Nous allons en plus les trier par nombre de branches.

Le principe du tri par comptage est de réaliser un 1er parcours pour déterminer combien il y a d'étoiles par nombre de branches, de réserver des espaces pour chaque catégorie d'étoile en fonction de leurs branches, puis, de réaliser un 2d passage pour placer les étoiles dans les espaces réservés.

Regardons et décrivons pour chaque diapositive, le déroulement pas à pas sur l'animation :

- 1 : on réserve l'espace de résultat pour toutes les étoiles
- 2 : on compte les étoiles par catégorie de branche
- 3 : on connaît déjà les formes qui seront à placer
- 4 : on réalise la somme cumulée de tous les sous-ensembles. Cette somme nous indique la position de départ de stockage de chaque catégorie (index en gras en haut de la diapositive).
- 5-6 : on considère ces sommes cumulées comme autant d'index pour placer les étoiles que l'on trouvera sur notre parcours. Pour cela, il faut ajouter l'index 0 pour la première catégorie.
- 7 : La 1re étoile est à 5 branches, on utilise donc le 3e index (pour les étoiles à 5 branches).
- 8 : On place la 1re et on incrémente le compteur à 5 branches. On peut examiner la 2e étoile qui est à 4 branches et qui devra donc aller dans la place libre la plus à gauche indiquée par le 2e index (pour les étoiles à 4 branches).
- 9 : toutes les étoiles rouges ont été placées dans les premiers espaces réservés.
- 10-12 : toutes les étoiles vertes sont placées dans les espaces suivants correspondant à leur nombre de branches.
- 13-14 : toutes les étoiles bleues sont placées dans les espaces restants correspondant à leur nombre de branches.

On peut remarquer que tous les index font référence au début de l'espace suivant. Il suffit d'ajouter un index 0 en tête pour se retrouver dans l'état d'origine (avant le placement des étoiles). Nous utiliserons cette propriété pour le chaînage des fils (racine vers feuilles) de notre algorithme.

Ce tri s'accompagne d'une propriété remarquable : il ne modifie pas les tris précédents. Les étoiles étaient triées par couleur et nous les avons placées par nombre de branches en les prenant dans l'ordre des couleurs. Nous obtenons donc des étoiles triées par nombre de branches et dans chaque catégorie les étoiles restent classées par couleur.

7.2 Création des voisins

La création des voisins opère de ce principe de tri par comptage. Vous trouverez les détails dans la classe `GraphWalker` qui se trouve dans le répertoire `tree`. La classe fournit plusieurs méthodes, dont certaines mentionnées dans la section "4.3 GraphWalker".

- `getEdges` : remplit un tableau de voisins avec le poids calculé suivant la fonction fournie en paramètre
- `getCountingSortedEdges` : fait la même chose en réalisant un tri par comptage

Le tableau des voisins doit être préalablement alloué en fonction du nombre de voisins.

Nous obtenons ce genre d'utilisation :

```
Size size = option.size;
Raster raster (size);
// ... read pixels
GraphWalker graphWalker (size);

vectorEdge edges (graphWalker.edgeMaxCount ());

graphWalker.getCountingSortedEdges (&edges [0], MinWeight (raster));
```

List. 16: Utilisation de `getCountingSortedEdges`

Si vous lisez attentivement le code de la méthode `getCountingSortedEdges`, vous retrouverez les étapes décrites dans la section "7.1 Tri par comptage".

7.3 Chaînage vers les parents

Pour réaliser le chaînage des feuilles vers la racine, nous allons mettre en œuvre la méthode présentée dans la section "3.3 Principe de construction d'arbre".

La figure 17 prend comme exemple la matrice de pixels déjà utilisée dans la section "5 Représentation arborescente via des tableaux". Une animation correspondant à cet exemple est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-4-details-parents.gif`.

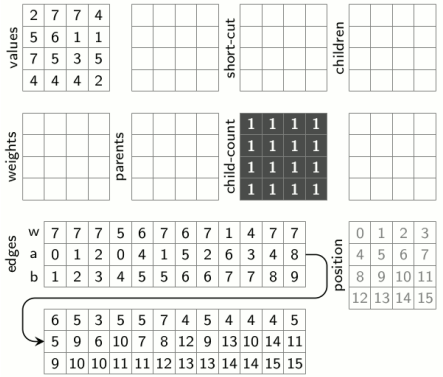


Fig. 17: M4105C-4-details-parents

Vous remarquerez qu'au début seul le tableau des valeurs de pixels contient des chiffres. Nous avons en plus initialisé à la valeur 1, tous les compteurs de fils des potentiels parents.

Nous ajoutons 3 autres tableaux :

- *short-cut* (en haut au milieu) qui mémorisera un raccourci vers le pixel maximum de la zone en cours de construction.
- *position* (en bas à droite) qui nous sert simplement d'aide pour repérer facilement un index dans une matrice de pixels.
- *edges* le tableau horizontal des voisins, si grand qu'on a du le séparer en deux (une flèche indiquant la suite de la 1re partie vers la 2de).

Le tableau des voisins est constitué de triplets (w, a, b).

- w est le poids (valeur maximum des pixels a et b)
- a est la position dans la matrice du 1er voisin
- b est la position dans la matrice du 2d voisin

Voici quelques étapes remarquables dans les diapositives de cette animation.

- 2 : les voisins sont identifiés et placés dans l'ordre de leur analyse, à savoir tous les voisins horizontaux dans le sens de lecture classique (du coin haut gauche vers le coin bas droit), puis tous les voisins verticaux en utilisant le même principe de lecture.
- 3 : les voisins sont triés par ordre croissant de poids (en conservant leur précédence spatiale).
  - Puis on traite le 1er couple de voisin de poids 1 (en rouge).
  - Comme ils n'ont pas de parenté **et** qu'ils ont le même poids, on choisi celui qui possède le plus de fils comme pixel maître. Ici, comme ils en ont autant, on tire au hasard celui qui sera pixels maître. Ce sera pour nous le représentant du nœud père dans l'image. On met à jour les flèches qui partent des deux voisins vers ce pixel maître.
  - C'est la première fois que ce pixel est maître, donc on crée son père (poids 1 dans le tableau *weights* et case rouge dans le tableau *parents*).
  - On ajoute aux pixels maîtres tous les fils de l'autre dans le tableau *child-count*.
  - On fait référence à ce nouveau père en plaçant son index (0) dans les deux cases correspondantes le tableau *parents* des pixels.
- 4 : On traite maintenant la série des voisins de poids 3 (en jaune).
  - Comme ils n'ont pas de parenté **et** qu'ils ne sont pas de même poids, on désigne d'office celui qui a le plus fort poids comme pixel maître. On met à jour les flèches qui partent de ces deux voisins vers ce pixels maître.
  - C'est la première fois que ce pixel est maître, donc on crée son père (poids 3 dans le tableau *weights* et case jaune dans le tableau *parents*).
  - On incrémente seulement de 1 la case du parent dans le tableau *child-count* (car ce parent n'hérite que d'un nouveau fils : **le** pixel maître du niveau inférieur).

- On fait référence à ce nouveau père en plaçant son index (1) dans les deux cases correspondantes du tableau *parents* des pixels et du parent du pixel maître précédent.

- 7 : Cette étape est un exemple de connexion de deux zones faisant partie du même nœud.
- 27 : C'est à ce moment que l'on identifie une zone à fusionner.
- 28 : Cette étape donne le résultat après fusion.

Nous aborderons la question de la fusion des nœuds "frères" faisant partie de la même zone par la suite. Nous nous proposons dans un premier temps de réaliser les étapes 1 à 26, ce qui correspond à la plus grande partie du traitement.

Une autre façon de décrire ce traitement est de suivre l'algorithme 1.

```
procedure BUILDPARENTS(GraphWalker graphWalker, MinWeight weightClass)
    size ← graphWalker.size
    compCount ← 0
    vectorEdgeedges(graphWalker.edgeMaxCount())
    edgeCount ← graphWalker.getCountingSortedEdges(edges, weightClass)
    for edgeId ← 0, edgeCount − 1 do
        curEdge ← edges[edgeId]
        pa ← point2idx(size, curEdge.points[0])
        pb ← point2idx(size, curEdge.points[1])
        la ← leaders.find(pa)
        lb ← leaders.find(pb)
        ra ← leafParents[la]
        rb ← leafParents[lb]
        if la = lb then
            continue
        if ra = DimImg_MAX then
            swap(la, lb)
            swap(ra, rb)
        leader ← DimImg_MAX
        if ra = DimImg_MAX then
            createParent(compCount, curEdge.weight, leafParents[la], leafParents[lb])
            if weightClass.isWeightInf(weightClass.getWeight(la), weightClass.getWeight(lb)) then
                swap(la, lb)
            leader ← la
        else if rb = DimImg_MAX then
            if curEdge.weight = compWeights[ra] then
                addChild(ra, leafParents[lb])
                leader ← la
            else
                createParent(compCount, curEdge.weight, compParents[ra], leafParents[lb])
                leader ← lb
        else if ra = rb then
            leader ← lb
        else if compWeights[ra] = compWeights[rb] then
            if childCountRec[ra] < childCountRec[rb] then
                swap(la, lb)
                swap(ra, rb)
            addChildren(ra, rb)
            leader ← la
        else
            if weightClass.isWeightInf(compWeights[ra], compWeights[rb]) then
                swap(la, lb)
                swap(ra, rb)
            leader ← la
            if compWeights[ra] = curEdge.weight then
                addChild(ra, compParents[rb])
            else
                createParent(compCount, curEdge.weight, compParents[ra], compParents[rb])
        leaders.link(pa, leader)
        leaders.link(pb, leader)
    setRoot(compCount)
end procedure
```

Algo. 1 : Chaînage feuilles vers racine

8 TP4 : Chaînage vers la racine

Les objectifs de ce TP sont de :

- contourner un problème d'insertion
- comprendre un algorithme linéaire
- mettre en place le chaînage feuilles vers racine

8.1 Insertion sans coût

Comme vous l'avez compris le tri par insertion se fait en deux phases. Dans la 1re phase on crée une suite croissante d'entiers (somme cumulée). Dans la 2de phase on doit insérer en tête un index de valeur 0 avant de parcourir les éléments à placer.

Prenons un instant pour comprendre le coût de cette insertion. Partons du code suivant :

```
1  const int n = 10;
   int tab1 [n];
   int tab2 [n+1];

5  for (int i = 0; i < n; ++i)
     tab [i] = i;

   for (int i = 0; i < n; ++i)
     tab2 [i+1] = tab[i];
10 tab2 [0] = 0;
```

List. 17: Insertion avec copie des éléments

Question : Quelle est la complexité de cet algorithme d'insertion en fonction de n ? Serait-il différent pour insérer 10 cases en tête du tableau `tab1` ? Peut-on faire plus rapide ?

Réponse : ✍

Et si nous faisions l'insertion de 10 cases au tableau en une seule instruction. . . ?

Vous le savez, en C++ un tableau est un pointeur sur le 1er élément. Observez le programme suivant :

```
1  const int n = 10;
   int tab2 [n+1];
   int *tab1 = tab2+1;

5  for (int i = 0; i < n; ++i)
     tab1 [i] = i;

   tab2 [0] = 0;
```

List. 18: Insertion avec décalage de pointeur

L'intérêt est en fait de remplir directement `tab2` au travers de `tab1`.

Le second intérêt est de ne pas avoir d'allocation mémoire. Cela peut paraître négligeable. Cependant, lorsque nous traitons des images dont les pixels prennent des valeurs sur 16 bits, le tableau de fréquences des poids des voisins est de 256 Mo. Créer une second tableau, juste pour ajouter un 0 en tête n'est pas fair-play.

8.2 Surcharge de `buildParents`

La librairie `libmerciol.a` possède déjà une classe `ArrayTree` avec la méthode `build` suivante :

```
1  void
   ArrayTree::build (const GraphWalker &graphWalker, const MinWeight &weightClass) {
     DEF_LOG ("ArrayTree::build", "");
     buildInit (graphWalker);
5    buildParents (graphWalker, weightClass);
     compressParents ();
     buildChildren ();
     buildFinish ();
   }
```

List. 19: Méthode `ArrayTree::build`

Vous avez déjà créé votre classe dérivée.

Question : Réécrivez la méthode virtuelle `build` dans votre classe dérivée suivant l'algorithme présenté dans la section précédente. La première ligne de code doit être une trace montrant que vous passez par votre méthode.

Réponse : ✍

Question : Vérifiez que la méthode de la classe de base n'est pas appelée en lançant votre test avec l'option `-debug`.

Réponse : ✍

Pour tester que le résultat de votre classe est identique à celui attendu vous pouvez vous inspirer du code suivant :

```
ArrayTree arrayTree;
arrayTree.build (GraphWalker (size), MinWeight (raster));

MaClassDeriveArrayTree arrayTree2;
arrayTree2.build (GraphWalker (size), MinWeight (raster));

cerr << "res " << arrayTree.compareTo (arrayTree2) << endl;
```

List. 20: Test de validité de *build*

Question : Vérifiez que les résultats sont identiques avec différents jeux de tests (en modifiant les valeurs dans `Raster`).

Réponse : ✍

Question : Comparez les performances avec la méthode de la classe de base, comme dans le TP1.

Réponse : ✍

Page de notes personnelles : ✍

9 Présentation de la compression

Cette section est peut-être la plus délicate. Si vous buttez sur ce passage vous pouvez poursuivre avec le chaînage inverse (racine vers feuilles) et y revenir plus tard.

La compression se réalise en 2 temps :

- Le choix de **nouveaux index** plus compacts pour les nœuds
- La modification de ces nouveaux index. Cette dernière se décline de nouveau en 2 temps :
  - **changement des valeurs** d'index dans les pixels et les nœuds
  - **permutation des nœuds** pour qu'ils soient à la bonne place

Nous allons aborder la possibilité de parallélisation dans la section "15 Parallélisation". Une partie de la compression ne peut pas être parallélisée car nous souhaitons pour tous les nœuds un index unique et croissant avec leur poids.

En revanche, nous donnons ici la solution générale permettant de réaliser cette compression après la construction de plusieurs arbres correspondant chacun à une tuile de l'image.

Nous n'allons pas reprendre la même matrice que précédemment. Mais nous partons d'une image de 12 pixels (4x3) comprenant 3 tuiles. Chaque tuile a produit un arbre, dont nous représentons les parents dans la couleur correspondant à leur tuile.

Nous partons d'une situation où les nœuds sont déjà triés par poids croissant dans les espaces rouges, verts et bleus. Nous avions réservé assez de places pour les nœuds, sachant qu'il ne peut pas y en avoir plus que de pixels. Enfin, nous avons réalisé un "tissage" avec les voisins frontaliers des tuiles pour "reconnecter" le chaînage entre les arbres des différentes tuiles.

Opération de tissage

Ce document ne présente pas la manière de "tisser" les arbres. L'opération peut amener à supprimer des nœuds ou en créer de nouveaux. Dans le cas de création, les nœuds seront placés dans les trous laissés libres entre les parents des différentes tuiles. Voir [5] pour plus de détails.

9.1 Nouveaux index

Voici la légende des figures que nous utiliserons dans cette section :

- W : poids du nœud
- P : parent du nœud
- CC : compteur de fils
- N : nouvel index du nœud

Nous commençons par le choix de **nouveaux index** avec la figure 18. Une animation est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-6-new-index.gif`. L'algorithme consistera à disposer d'autant de pointeurs de "base" qu'il y a de tuiles. On commencera le traitement par le pointeur qui réfère un poids minimum (0 dans ce cas). On applique notre indexation (le numéro suivant d'un simple compteur). Puis on poursuivra avec ce pointeur de "base" tant que la tuile fournira une valeur de poids minimum. Ensuite, on passe au pointeur de base qui possède la même valeur de poids (i.e. 0). Si au bout d'un tour complet on revient à la même position, c'est qu'il n'y a plus de tuile de valeur 0. On passe alors à la valeur minimum suivante.

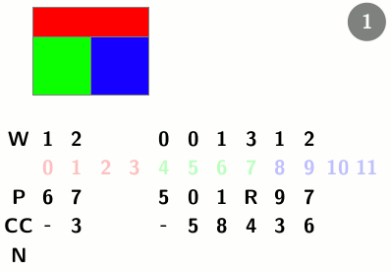


Fig. 18: M4105C-6-new-index

L'algorithme consiste à appliquer `updateNexIndex` successivement sur le plus

faible élément des listes de nœuds fournies.

```
callOnSortedSets(sizes, sortedParentsList, updateNewIdx(compIdx, lastIndex))
procedure updateNexIndex(DimImg curComp, DimImg lastIndex)
    if newCompIdx[curComp] ≠ DimImg_MAX then
        return
    top ← findCompMultiChild(curComp)
    if curComp ≠ top then
        newTopIdx ← newCompIdx[top]
        if newTopIdx = DimImg_MAX then
            newTopIdx ← newCompIdx[top] ← compCount++
        newTopChildCountRec ← childCountRec[top]
        newTopCompParent ← compParents[top]
        newTopWeight ← compWeights[top]
        for sibling ← curComp, top do without increment
            nextSibling ← compParents[sibling]
            newCompIdx[sibling] ← newTopIdx
            childCountRec[sibling] ← newTopChildCountRec
            compParents[sibling] ← newTopCompParent
            compWeights[sibling] ← newTopWeight
            sibling ← nextSibling
        return
    newCompIdx[curComp] ← compCount++
end procedure
```

Algo. 2 : Choix des nouveaux index

Comme nous traitons l'image en une fois (pas de traitement parallèle par tuile), `callOnSortedSets` revient à une simple boucle sur l'ensemble des nœuds existants.

Dans l'animation, voici quelques diapositives significatives :

- 3 : il y a 3 pointeurs de "base" : 0 pour rouge, 4 pour vert et 8 pour bleu
- 4 : le poids minimum est 0 pour le pointeur vert. Son nouvel index est le 1er donc la valeur 0.
- 5 : on met à jour tous les frères de ce nœud, qui sont obtenus en suivant le lien de parenté et qui reste dans la même zone (les nœuds 4 et 5 ont le même père d'index 0). On en profite pour copier le nombre de fils du frère le plus haut dans le chaînage (c'est lui qui connaît le nombre exact de fils dans la zone).
- 6 : on propage le nouvel index à tous les frères connus de cette lignée.
- 7 : comme le pointeur vert 5 est déjà traité on passe à 6. Le poids a changé, nous passons aux pointeurs de couleur suivant en mémorisant le poids minimum (i.e. 1). Après un tour complet, nous n'avons pas trouvé d'autre poids 0.
- 8 : Nous prenons un pointeur de "base" de poids minimum (nous avions noté que c'était la valeur 1). Ce sera le pointeur de "base" vert 6
- 18 : le traitement se termine lorsque nous n'avons plus de pointeur de "base"

9.2 Changement des valeurs

Nous disposons de nouveaux index. Il faut mettre à jour tous les parents des pixels et des nœuds. Ce traitement peut être réalisé en parallèle. Il faut simplement veiller à ce qu'il ne soit pas fait 2 fois.

Fig. 19: M4105C-7-update-index

La figure 19 ne permet de montrer que la mise à jour des index des nœuds. L'animation correspondante est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-7-update-index.gif`.

L'algorithme 3 réalise cette mise à jour.

Il reste maintenant à réaliser la dernière étape qui matérialise la compression. Il s'agit de remettre les nœuds dans l'ordre défini par les nouveaux index.

```
procedure updateNexIndex(DimImg maxNode)
    foreach leaf ∈ pixels do in parallel
        old ← leafParents[leaf]
        if old ≠ DimImg_MAX then
            leafParents[leaf] ← newCompIdx[old]
    foreach node ∈ nodes do in parallel
        old ← compParents[node]
        if old ≠ DimImg_MAX then
            compParents[node] ← newCompIdx[old]
end procedure
```

Algo. 3 : Changement des valeurs

9.3 Permutation des nœuds

Nous avons profité du moment de l'indexation pour recopier les informations qui pouvaient différer (nombre de fils et nouvel index) entre un frère et ses aînés. De sorte que nous disposons à cette étape d'un ensemble de nœuds non nécessairement ordonnés. Cependant, nous avons la garantie que tous les frères d'une même zone sont interchangeables.

Le procédé va consister en un balayage de tous les nœuds pour les placer à leur juste place. Nous souhaitons le faire sans ajout de mémoire supplémentaire. L'activité principale va consister à prendre une position (en commençant par la première) et tant que le nœud qui s'y trouve n'est pas à sa place, le permuter avec celui qui se trouve à la sienne.

Il faut prendre garde aux nœuds vides et aux doublons. Dans le cas où on ne peut pas agir, on passe seimplement à la place suivante. Le problème rencontré trouvera sa solution de lui-même par la suite.

L'ensemble du processus schématisé dans la figure 20 est de nouveau décrit dans une animation qui est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-8-compress.gif`.

Fig. 20: M4105C-8-compress

Dans l'animation, voici quelques diapositives significatives :

- 1 : Le nœud à cette position a comme nouvel index 1. Il doit être déplacé.
- 2 : On permute donc les nœuds O et 1. Ce nouveau nœud a comme nouvel index 3. Il doit être déplacé.
- 4 : On permute les nœuds 0 et 3. Ce nouveau nœuds est vide. Il faut passer au suivant.
- 6 : Ce nœud est déjà en place, on passe au suivant.
- 9 : Le nœud à cette position a comme nouvel index 0. Il doit être déplacé.
- 12 : Le nœud à cette position a comme nouvel index 0. Il devrait être déplacé, mais il y a déjà un frère qui s'y trouve, on passe donc au suivant.
- 19 : L'index 9 est la valeur la plus grande dans l'ancienne numérotation. Il n'y aura plus aucune permutation ensuite.

L'algorithme s'écrit simplement avec une boucle (cf algorithm 4).

```
procedure swapNodes(DimImg maxNode)
    for curComp ← 0, maxNode do without increment
        newIdxComp ← newCompIdx[curComp]
        if newIdxComp = curComp or newIdxComp = DimImg_MAX or
            newCompIdx[newIdxComp] = newIdxComp then
            ++curComp
            continue
        swap(compParents[curComp], compParents[newIdxComp])
        swap(compWeights[curComp], compWeights[newIdxComp])
        swap(childCountRec[curComp], childCountRec[newIdxComp])
        swap(newCompIdx[curComp], newCompIdx[newIdxComp])
end procedure
```

Algo. 4 : Permutation des nœuds

Réfléchissons sur la complexité de cette fonction. Chaque permutation de nœuds et définitive, leur occurrence est inférieure à n. Chaque analyse d'une position amène : soit à une permutation (dans la limite de n) soit à passer au suivant. La complexité est donc en $O(n)$.

10 TP5 : Compression

Les objectifs de ce TP sont de :

- comprendre et documenter une fonction de réordonnancement
- visualiser les fusions

Comme pour le TP précédent, vous poursuivez avec la classe dérivée que vous avez créée.

Question : Réécrivez la méthode virtuelle `compressParents` dans votre classe dérivée suivant l'algorithme présenté dans la section précédente. La première ligne de code doit être une trace montrant que vous passez par votre méthode.

Réponse : ✍

Question : Faîtes les modifications pour tracer les situations de fusion.

Réponse : ✍

Page de notes personnelles : ✍

11 Chaînage vers les feuilles

La disposition du chaînage complet des feuilles vers la racine est suffisante pour qualifier la structure d'arbre. Cela permet de dérouler des algorithmes de parcours "bottom-up" des feuilles vers la racine. C'est ce parcours que nous employons pour la production de certains attributs, comme le calcul du nombre de pixels couverte par un nœud de l'arbre.

Cependant, si nous souhaitons profiter d'une architecture multi-cœur, nous perdrions le profit du traitement parallèle par un étranglement d'exclusion mutuelle de 2 fils cherchant à mettre à jour un même parent. On préférera traiter simultanément tous les pères de même niveau qui interrogeront leurs fils, sans aucun risque d'accès concurrent.

Avec la figure 21, nous allons partir de la situation produite par le chaînage vers la racine. Une animation est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-5-details-children.gif`. La fin de l'animation aboutit à l'exemple présenté à la section "5 Représentation arborescente via des tableaux".

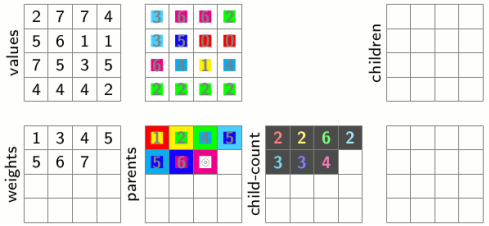


Fig. 21: M4105C-5-details-children

L'objet de ce chaînage sera de remplir l'ensemble des deux tableaux de droite nommé *Children*. Il s'agit en réalité d'un seul et unique tableau qui a été coupé en deux pour des raisons esthétiques.

La stratégie de construction de ce tableau relève du même principe que le tri par comptage. On commence par compter les fils de chaque nœud. Puis, on réserve l'espace des ensembles des fils. Et enfin, on parcourt tous les éléments pour les placer judicieusement.

Il se trouve que le comptage de fils a déjà été effectué lors du chaînage vers les parents pour des raisons de minimisation du nombre de nœuds frères. L'algorithme 5 s'en trouve simplié.

procedure BUILDCHILDREN
  $compCount \leftarrow getCompCount()$
  $partial\_sum(childCountRec, childCountRec + compCount, childCountRec)$
  $childGetOrder \leftarrow childCount + 1$
  for $i \leftarrow 0, nodeCount - 1$ **do**
    if $leafParents[i] = DimImg\_MAX$ **then**
      continue
    $children[childGetOrder[leafParents[i]] + +] \leftarrow i$
end procedure

Algo. 5 : Chaînage racine vers feuilles

Voici quelques diapositives significatives de l'algorithme :

- 2 : La somme cumulée des compteurs indique la limite des espaces des fils pour chaque nœud. En ajoutant l'index 0, on obtient une référence vers le début de chaque espace des fils (cf. la section "7.1 Tri par comptage").
- 4 : Le 1er fils (index 0) fait référence au parent 3. L'index *child-count* de ce parent fait référence à la 11e case (index 10). On place dans cette dernière case la référence de l'élément traité : 0
- 5 : Une fois placé l'élément précédent, on incrémente l'index de position, qui passe de la valeur 10 à 11. Et on regarde l'élément suivant (index 1) qui fait référence au parent 6.
- 26 : Le traitement se termine en arrivant sur le dernier parent (la racine de l'arbre). Tous les index de zones de fils pointent sur les fins de zone.
- 27 : Il reste à insérer un index 0 en tête des index pour retrouver la désignation complète des frontières entre chaque espaces de fils.

Ce chaînage possède des propriétés très particulières comme indiqué en la section "5 Représentation arborescente via des tableaux".

- Les fils sont classés par ordre croissant de position de pixels, puis par ordre croissant de poids des parents.
- On peut rechercher dans le tableau *weights* tous les parents de même poids, puis à l'aide des index de *child-count* appliquer un traitement en parallèle, car nous avons l'assurance qu'il ne provoquera pas de conflit d'accès.

Comme vous le constatez en regardant l'animation, chaque pixel est observé une fois et une seule (n) et chaque nœud est observé une fois et une seule

($< n$). La somme cumulée dépend du nombre de nœuds ($< n$). L'insertion d'index à 0 dans un tableau ne demande qu'une seule instruction. L'algorithme est en $O(n)$

12 TP6 : Chaînage vers les feuilles

Les objectifs de ce TP sont de :

- comprendre un algorithme linéaire
- mettre en place le chaînage vers les feuilles

Comme vous l'avez remarqué, le placement des fils est très semblable à celui d'un tri par comptage. Vous allez devoir le réaliser par vous-même.

La librairie `libmerciol.a` possède déjà une classe `ArrayTree` avec la méthode virtuelle `buildChildren`.

Vous avez déjà créé votre classe dérivée.

Question : Réécrivez la méthode virtuelle `buildChildren` dans votre classe dérivée suivant l'algorithme présenté dans la section précédente. La première ligne de code doit être une trace montrant que vous passez par votre méthode.

Réponse : ✍

Question : Vérifiez que la méthode de la classe de base n'est pas appelée en lançant votre test avec l'option `-debug`.

Réponse : ✍

Pour tester que le résultat de votre classe est identique à celui attendu vous pouvez vous inspirer du code suivant :

```
1  ArrayTree arrayTree;
   arrayTree.build (GraphWalker (size), MinWeight (raster));

   MaClassDeriveArrayTree arrayTree2;
5  arrayTree2.build (GraphWalker (size), MinWeight (raster));

   cerr << "res " << arrayTree.compareTo (arrayTree2) << endl;
```

List. 21: Test de validité de *buildChildren*

Question : Vérifiez que les résultats sont identiques avec différents jeux de tests (en modifiant les valeurs dans `Raster`).

Réponse : ✍

Question : Comparez les performances avec la méthode de la classe de base, comme dans le TP1.

Réponse : ✍

Page de notes personnelles : ✍

13 Exploitation d'un arbre

Avant de parler du traitement de l'élagage des arbres, il nous faut choisir un critère d'élagage. Nous allons donc commencer par créer des attributs associés aux nœuds de l'arbre.

13.1 Paradoxe des anniversaires

Nous abordons donc la question du calcul d'attributs. Comme nous souhaitons le faire de façon linéaire et efficace (profiter de tous les cœurs de notre matériel), regardons si on peut réaliser un traitement en parallèle. Le risque pour nous étant de manipuler simultanément un même nœud par 2 traitements distincts. Pour cela, faisons un détour du côté des problèmes de collision d'informations.

Un problème connu concernant les collisions se nomme le "paradoxe des anniversaires". Si vous êtes seul dans une pièce et que vous cherchiez à savoir si une autre personne que vous a le même jour anniversaire dans l'année, vous risquez d'être déçu. Si, en revanche, vous vous trouvez dans une pièce où se trouve 365 personnes (disons que l'on ne compte pas les années bissextiles), et que vous vous posez la question de savoir s'il existe 2 personnes ayant le même jour anniversaire, vous allez sûrement trouver une paire.

Fig. 22: Paradoxe des anniversaires

La question est : à partir de combien de personnes dans une pièce pouvez-vous parier qu'il existe une paire qui soit née le même jour ?

La probabilité qu'une telle coïncidence survienne est de : $pc(n) = \frac{365!}{365-n!} \cdot \frac{1}{365^n}$.

La probabilité inverse (qu'il n'y ait pas de coïncidence) est $\overline{pc(n)} = 1 - pc(n)$.

Ces deux probabilités sont représenté sur la figure **??**.

Dans notre situation, nous gérons des images satellite de plusieurs millions de pixels qui prennent des valeurs sur 16 bits (65 536 valeurs possibles). Certes une année de 65 536 jours, c'est long. Mais, une salle contenant plusieurs millions de personnes est encore plus impressionnante.

On peut dire qu'à chaque niveau d'un arbre (valeur possible des pixels) correspondant à une image de 10 millions de pixels, le risque de trouver des nœuds de même valeur est très élevé. Par ailleurs, Il est en moyenne de 150 nœuds à chaque fois. On comprends, qu'il sera possible d'exploiter tous les cœurs d'une machine pour le traitement de chaque niveau de valeur de pixel.

13.2 Calcul de surface

Le calcul du nombre de pixels référencés par un nœud (donc la surface d'une zone) est simple.

- pour les pixels, la surface est de 1 (traitement qui se réalise facilement sur une partie de l'image en proportion par chaque cœur)
- pour les nœuds, la surface est la somme des surfaces de tous ces fils : pixels comme autres nœuds (traitement qui peut se réaliser en parallèle comme indiqué à la section précédente).

Ce qui revient à l'algorithme suivant :

```
procedure computeAreas
    alloc(areas[leafCount + nodeCount])
    foreach leaf ∈ pixels do in parallel
|     area[leaf] ← 0
    foreach node ∈ nodes do in parallel for same weight
|     area ← 0
|     for child ← childCount[node], childCount[node + 1] do
|         area ← area + areas[children[child]]
|     compAreas[node] ← area
end procedure
```

Algo. 6 : Calcul de surface

13.3 Élagage d'arbre

L'élagage va consister à parcourir l'arbre à partir des feuilles en considérant 2 valeurs : un seuil et un plafond. En parcourant une branche, depuis sa feuille vers la racine, on ne considérera que les nœuds dont la taille est entre ces deux valeurs. On prendra comme poids celui du nœud le plus élevé de cet intervalle.

procedure CUT(DimImg $thresholdA$, DimImg $thresholdB$, Raster $outputRaster$)
  $rootId \leftarrow arrayTree.getCompCount() - 1$
  foreach $leafId \in pixelsinarrayTree$ **do**
    $nodeValueA \leftarrow raster.getPixels()[leafId]$
    $size \leftarrow 1$
    $nextParentId \leftarrow arrayTree.getLeafParent(leafId)$
    if $nodeValueA = arrayTree.getWeight(nextParentId)$ **then**
      $size \leftarrow areas[nextParentId]$
      $nextParentId \leftarrow arrayTree.getCompParent(nextParentId)$
    while $nodeValueA < thresholdA$ **and** $nextParentId < rootId$ **do**
      $parentId \leftarrow nextParentId$
      $nextParentId \leftarrow arrayTree.getLeafParent(nextParentId)$
      $nodeValueA \leftarrow arrayTree.getWeight(nextParentId)$
    $nodeValueB \leftarrow nodeValueA$
    while $nodeValueB < thresholdB$ **and** $nextParentId < rootId$ **do**
      $parentId \leftarrow nextParentId$
      $nextParentId \leftarrow arrayTree.getLeafParent(nextParentId)$
      $nodeValueB \leftarrow (Pixel)arrayTree.getWeight(nextParentId)$
    $outputRaster.getPixels()[leafId] \leftarrow nodeValueA! = nodeValueB?nodeValueA : 0$
end procedure

Algo. 7 : Calcule de surface

14 TP7 : Parcours d'arbres

Les objectifs de ce TP sont de :

- réaliser des parcours d'arbres en temps linéaire
- mesurer ce qui peut être traité en parallèle

Ce TP propose des exercices de parcours d'arbre sur les structures déjà existantes.

Une nouvelle classe vous est offerte : `AreaAttr` qui sert à gérer les attributs de surface.

Dans cet exercice, il n'y aura pas de remplissage des feuilles à 1. Mais nous allons simplement vérifier que le fils est :

- soit une feuille : auquel cas on incrémente la valeur de surface de 1
- soit un nœud (donc non feuille) : auquel cas on ajoutera la surface déjà calculée pour ce fils

La librairie `libmerciol.a` possède déjà une classe `AreaAttr`. Vous avez à la dériver pour vous exercer à reproduire ses effets.

14.1 Surface

Question : Réécrivez la méthode virtuelle `update` dans votre classe dérivée suivant l'algorithme de mise à jour des surfaces des nœuds. La première ligne de code doit être une trace montrant que vous passez par votre méthode. **Attention, le nombre d'attributs est le même que le nombre de nœuds** (vous ne gérez pas les pixels).

Réponse : ✍

Question : Vérifiez que la méthode de la classe de base n'est pas appelée en lançant votre test avec l'option `-debug`.

Réponse : ✍

Testez que le résultat de votre classe est identique à celui attendu. Pour cela, comparez les valeurs des attributs produits avec ou sans votre classe dérivée.

Question : Vérifiez que les résultats sont identiques avec différents jeux de test (en modifiant les valeurs dans `Raster`).

Réponse : ✍

Question : Comparez les performances avec la méthode de la classe de base, comme dans le TP1.

Réponse : ✍

14.2 Élagage

Question : Réécrivez la méthode virtuelle `cut` dans votre classe dérivée suivant l'algorithme de la section précédente. La première ligne de code doit être une trace montrant que vous passez par votre méthode.

Réponse : ✍

Question : Vérifiez que la méthode de la classe de base n'est pas appelée en lançant votre test avec l'option `-debug`.

Réponse : ✍

Question : Vérifiez que les résultats sont identiques avec différents jeux de test (en modifiant les valeurs dans `Raster`).

Réponse : ✍

Question : Comparez les performances avec la méthode de la classe de base, comme dans le TP1.

Réponse : ✍

Page de notes personnelles : ✍

15 Parallélisation

Pour aller plus loin...

Le propos de ce module est d'aborder différents algorithmes linéaires. Mais nous avons déjà fait allusion à un autre levier d'optimisation de temps de calcul avec l'usage de traitement parallèle.

15.1 Limite d'application

Il faut prendre conscience que le traitement en parallèle d'une activité ne peut qu'exceptionnellement réduire le nombre d'instructions à réaliser.

Ce peut être le cas lorsque vous devez explorer plusieurs possibilités et que la solution se trouve être dans le chemin le plus court. Imaginons que nous devions explorer 4 labyrinthes et qu'un seul possède une solution. Les 3 autres disposent de cycle (boucles sans fin). Si vous commencez l'exploration d'un labyrinthe sans solution, vous tomberez dans une boucle infinie et ne regarderez jamais les autres.

Le traitement en parallèle des 4 problèmes aboutira à une solution pour un des labyrinthes. C'est à ce moment que vous devez prendre la décision d'interrompre les autres tâches.

C'est uniquement dans ces situations que le traitement en parallèle permet d'économiser des traitements.

Cependant, si vous disposez d'un serveur avec 40 cœurs, vous pourriez espérer réduire votre temps de calcul de ce facteur. Est-ce possible ?

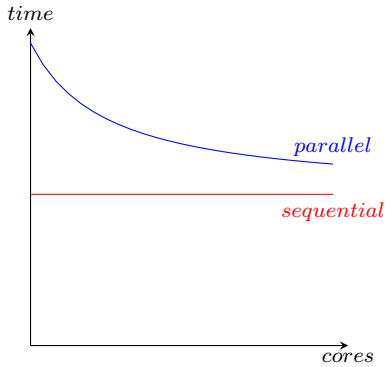


Fig. 23: Parallélisation

En fait, ce n'est pas si simple. Un traitement linéaire n'est pas naturellement parallélisable. Par exemple, si vous devez trier un ensemble, le tri rapide est parallélisable suivant la profondeur du découpage dichotomique. En revanche, le tri par fréquence ne l'est pas naturellement.

15.2 Traquer les contraintes séquentielles

Imaginez que vous disposiez d'un algorithme linéaire dont la moitié peut se faire en parallèle et l'autre non, la figure 23 montrerait l'évolution globale du temps de calcul. Elle est constituée de la somme du temps des traitements séquentiels et parallèles. Vous remarquez qu'en proportion, le temps d'exécution est rapidement entièrement consacré à la partie séquentielle (90 % à partir de 10 cœurs).

Vous comprenez pourquoi il est important de s'attaquer à paralléliser l'ensemble d'un l'algorithme.

Nous fournissons ici des indications sur des stratégies de parallélisation de la construction d'arbres que nous avons présentées plus haut.

15.3 Répartition géométrique

A partir du moment où le traitement n'est fonction que du nombre de pixels, il est incontournable de vouloir répartir ce traitement sur des surfaces identiques. Car tous les cœurs termineront leurs tâches au même moment, ce qui réduira le temps de latence.

Par ailleurs, il est souhaitable de réduire dans le même temps la longueur des frontières entre ces surfaces. C'est de cette longueur que dépendra le temps de fusion des résultats intermédiaires obtenus.

Voici un algorithme récursif qui répartit une surface en x cœurs.

```
procedure SHARE(int coreCount, rectangle tile, vectorRect tiles, vectorRect
boundaries, vectorBool verticalBoundaries)
    if coreCount < 2 or
       max(tile.width, tile.height) < 4 or
       min(tile.width, tile.height) < 3 then
        tiles.push_back(tile)
        return
    boolvertical ← tile.width > tile.height
    boolodd ← coreCount&0x1
    DimImgthin ← (vertical?tile.width : tile.height)/(odd?coreCount : 2)
    RecttileA(tile), tileB(tile)
    if vertical then
        tileA.width ← thin
        tileB.x ← tileB.x + thin
        tileB.width ← tileB.width − thin
    else
        tileA.height ← thin
        tileB.y ← tileB.y + thin
        tileB.height ← tileB.height − thin
    share((odd?1 : coreCount/2), tileA, tiles, boundaries, verticalBoundaries)
    boundaries.push_back(tileB)
    verticalBoundaries.push_back(vertical)
    share((odd?coreCount − 1 :
coreCount/2), tileB, tiles, boundaries, verticalBoundaries)
end procedure
```

Algo. 8 : Répartition géométrique

La figure 24 représente une animation qui est disponible dans votre répertoire `doc/images` ou pointée par `http://m4105.merciol.fr/_media/supports/m4105c-9-share.gif`.

L'animation présente 2 sortes de découpes :

- diapo 1,3,4 : le nombre de cœurs est pair ; le côté le plus grand est séparé en 2 parties identiques
- diapo 2 : le nombre de cœurs est impair ; le côté le plus long est séparé en 2 partie $1/n$ et $(n-1)/n$ (ici $n = 5$)

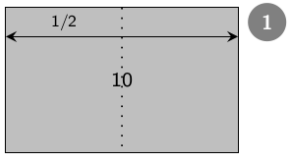


Fig. 24: M4105C-9-share

15.4 Utilisation des tâches Boost

De nombreuses briques logicielles ont été développées autour de C++. *Boost* vise à donner une cohérence à des fonctions susceptibles de rejoindre le langage.

Il existe dans *Boost* une section sur la gestion des tâches. Son usage est simple et se réduit principalement à deux fonctions :

- un constructeur qui prend en paramètre une fonction (l'équivalent de la méthode `run` dans la classe Java `Thread`)
- la méthode `join` qui permet d'attendre la fin d'une tâche.

Comme dans notre problème d'analyse d'images, nous avons souvent l'occasion d'avoir à lancer des traitements sur un sous-ensemble d'éléments semblables. Voici une fonction qui permet de répartir sur l'ensemble des cœurs d'une machine le traitement en fonction d'un segment d'index d'un ensemble.

```
procedure DEALTHREAD(DimImg maxId, int coreCount,
FunctThreadMinMax functThreadMinMax)
                                ▷ functThreadMinMax(threadId, minVal, maxVal)
    if !maxId or !coreCount then
        return
    if DimImg(coreCount) > maxId then
        coreCount ← maxId
        if coreCount = 1 then
            functThreadMinMax(0, 0, maxId)
            return
        vectorDimImgmaxIds ← getDealThreadBounds(maxId, coreCount)
        threadtasks[coreCount]
        for idCopyValInThread ← 0, coreCount − 1 do
            tasks[idCopyValInThread] =
        thread(functThreadMinMax(idCopyValInThread,
                [idCopyValInThread], maxIds[idCopyValInThread + 1]))
        for i ← 0, coreCount − 1 do
            tasks[i].join()
end procedure
```

Algo. 9 : Répartition de tâches

Nous avons également abordé dans la section "9 Présentation de la compression" l'utilisation d'un algorithme qui applique un traitement sur le plus petit

élément d'un ensemble de listes.

```
procedure callOnSortedSets(vectorDimImg sizes, WeightFunct
getWeight, CmpFunct isWeightInf, CallFunct callIdId)
    size ← sizes.size()
    if !size then
        return
    vectorDimImgvectIds(size, 0)
    vectorDimImgvectCounts(sizes)
    found ← false
    minVectIdx ← 0
    maxWeight ← 0
    for vectId ← 0, size − 1 do
        if !vectCounts[vectId] then
            continue
        tmpWeight ← getWeight(vectId, 0)
        if found and !isWeightInf(tmpWeight, maxWeight) then
            continue
        minVectIdx ← vectId
        maxWeight ← tmpWeight
        found ← true
    while found do
        found ← false
        nextMinVectIdx ← 0
        nextMaxWeight ← 0
        for vectId ← minVectIdx, do
            if vectCounts[vectId] then
                tmpWeight ← getWeight(vectId, vectIds[vectId])
                if !isWeightInf(maxWeight, tmpWeight) then
                    callIdId(vectId, vectIds[vectId])
                    + + vectIds[vectId]
                    − − vectCounts[vectId]
                    continue
                if !found or isWeightInf(tmpWeight, nextMaxWeight) then
                    nextMinVectIdx ← vectId
                    nextMaxWeight ← tmpWeight
                    found ← true
            vectId ← (vectId + 1)%size
        minVectIdx ← nextMinVectIdx
        maxWeight ← nextMaxWeight
end procedure
```

Algo. 10 : Traitement sur ensembles triés

15.5 Répartition algorithmique

Il y a plusieurs endroits où le traitement peut être réparti entre les cœurs d'une même machine :

- dans un premier temps, au moment du chaînage vers la racine. Au lieu de considérer l'image entière, on peut tuiler l'image avec des surfaces équivalentes (voir la section "15.3 Répartition géométrique"). Ce gain s'accompagne d'une contrepartie. Les tuiles ont fait apparaître des frontières dont les voisins n'ont pas contribué à la construction de l'arbre.

  Le nombre de voisins reste limité (d'une grandeur d'échelle de l'ordre de la racine carrée du nombre de pixels). D'expérience, nous avons pu constater que le plus efficace était de les traiter en un seul bloc et de les trier afin de minimiser les modifications de structure lors de la fusion des arbres. L'algorithme n'est pas fourni avec ce module, mais a déjà fait l'objet de publication [5].
- dans un second temps, dans la ré-indexation des nœuds au moment de l'élimination des "frères". En réalité, dans cet algorithme, seuls le choix de nouveaux index et le déplacement ne sont pas répartis sur les cœurs. En revanche, la mis à jour des nouveaux index l'est.
- Enfin, il est possible de répartir les usages qui seront faits de l'arbre sous forme de tableau. C'est ce que nous avons vu dans la section "14 TP7 : Parcours d'arbres". Le fait même que les parents soient triés par couches successives en fonction de leur poids, permet de traiter de manière horizontale la création d'attributs.

Il reste quelques points récalcitrants d'algorithmes séquentiels :

- le tri des voisins (le tri par comptage supporte mal l'accès concurrent aux compteurs)
- la construction des fils (pour les mêmes raisons)
- la ré-indexation des nœuds (qui a recours à un identifiant unique ordonné)

15.6 Performance

Il est temps d'aborder des mesures réelles d'utilisation.

Le traitement a été testé sur un ordinateur de 12 cœurs Xeon L5640 à 2.27 GHz disposant de 40 Go de mémoire vive. La configuration du serveur n'a pas été améliorée pour les besoins du test et tous les services habituels

(serveur ftp, web, ...) n'ont pas été arrêtés. Nous avons lancé la construction d'un arbre sur l'image 25 de 11 000x11 000 pixels.

L'algorithme en exécution séquentiel permet l'analyse de 4 000 000 de pixels par seconde, soit 1/4 de nanoseconde par pixel. Rappelons que ce temps est utilisé pour :

- rechercher les voisins et les trier
- construire les parents avec leur poids, leur nombre de fils et les trier
- déterminer les parents les plus représentatifs
- faire le chaînage vers les parents
- ré-indexer tous les parents pour fusionner les frères
- créer des fils et les trier
- créer des index d'ensembles de fils
- allouer la mémoire nécessaire à toutes ces structures

On remarque que l'algorithme est constant de part la platitude de la courbe bleue de la figure 26. Cette courbe indique le débit d'analyse en fonction de la taille de l'image.

Une 2de courbe au-dessus (en rouge), correspond au même traitement avec 12 cœurs. De nouveaux, elle souligne le caractère linéaire de l'algorithme avec une moyenne de 10 000 000 pixels par seconde.

On peut remarquer que le débit n'est pas multiplié par 12. Cela provient du phénomène développé au début de cette section. Les portions séquentielles de l'algorithme à la division du temps de calcul.

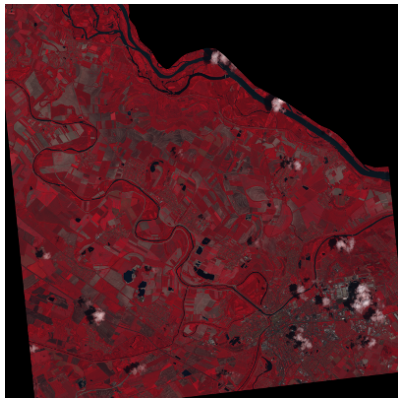


Fig. 25: Image satellite PAN

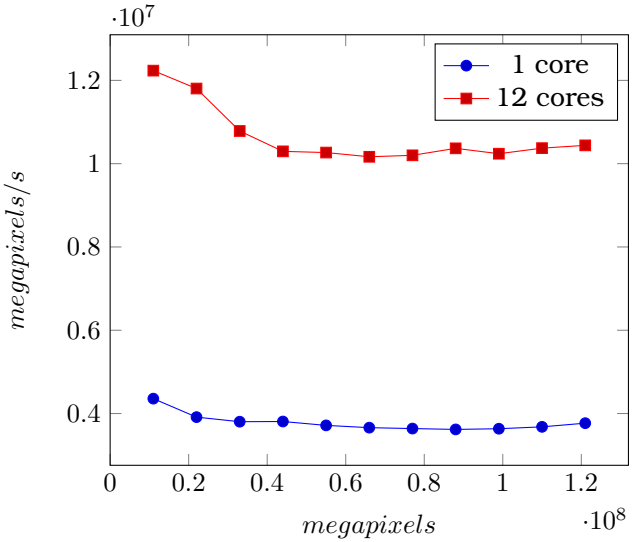


Fig. 26: Débit en fonction de la taille d'image

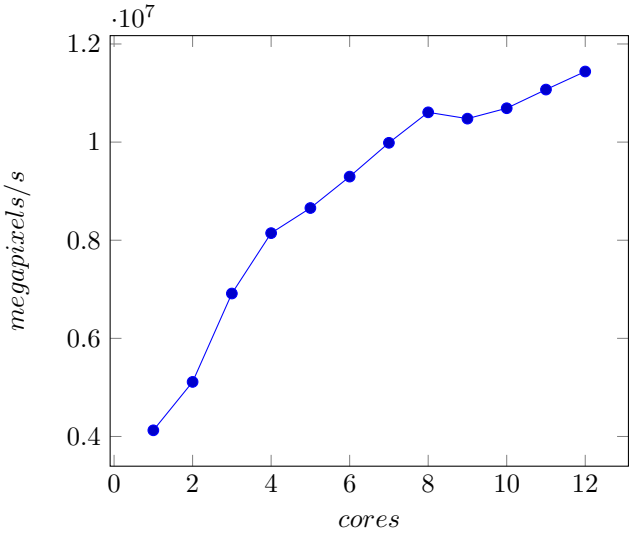


Fig. 27: Débit en fonctiondu du nombre de cœurs

16 Conclusion

Nous venons d'aborder plusieurs algorithmes efficaces en prenant comme support la construction l'analyse d'images. Les mécanismes que nous avons mis en avant auraient pu se retrouver dans d'autres situations. Cependant, cela nous permet de montrer la pertinence de leurs existences dans des utilisations des plus récentes et des plus avancées.

16.1 Choix de structures

Le choix de la structure est primordiale pour minimiser les traitements.

Il y a fréquemment des choix qui vous sont offerts entre mémoire et calcul. Si des traitements sont coûteux, vous pouvez chercher à pré-calculer certains éléments.

Par exemple, avec la suite de Fibonacci on peut mémoriser les 100 premières valeurs. Après une phase d'initialisation, la réponse sera instantanée pour les valeurs inférieures à 100. Et, vous gagnerez du temps à chaque appel supérieur. On peut également adopter une attitude intermédiaire, qui consiste à ne pas réaliser de pré-calculs, mais de mémoriser chaque fois qu'un traitement est demandé.

L'inconvénient de ces approches et qu'elles consomment de la mémoire. Il faut parfois les adosser à des mécanismes de nettoyage périodique.

Dans le cas du tri par comptage, une mémoire temporaire n'est utilisée que peu de temps. En revanche, sa taille dépend de l'espace des valeurs. C'est possible sur une machine "sérieuse" d'aller jusqu'à des valeurs sur 16 bits. Au delà, il faut vraiment se poser des questions.

Nous avons montré dans ce module le choix de structures se limitant exactement aux informations nécessaires. C'est à dire que nous n'avons délégué aucune gestion de structures à des librairies ou outils standards de type : vecteur, liste chaînée, pile, ...

La réduction de mémoire dans ce cas s'accompagne d'une réduction de consommation de mémoire. En effet, les structures dynamiques que nous venons de citer, s'accompagnent d'un coût d'appel au système d'exploitation pour allouer et libérer régulièrement des structures sous-jacentes masquées.

16.2 Analyse correcte du problème

L'optimisation revient parfois à une simple analyse du problème pour se rendre compte qu'il n'y a pas de problème.

Dans plusieurs algorithmes vus dans ce module, nous avons eu à utiliser des compteurs cumulés et à devoir insérer à plusieurs reprises des valeurs 0 en tête.

La démarche naturelle serait de créer une nouvelle donnée et de recopier les n éléments de la 1^{re} sur la 2^{de}. Parfois, on en oublie de vérifier si oui ou non la 1^{re} est encore utilisée par la suite. On vous apprend à ne pas réutiliser des variables pour éviter les erreurs par effet de bord de l'une sur l'autre. Ici, la situation est différente. Les différents pointeurs sur un même tableau sont clairement identifiés et ne correspond pas à un manque d'imagination de noms de variables.

16.3 Le comptage

Il y a une récurrence dans les algorithmes présentés dans ce module : le comptage. Énumérer des éléments permet d'anticiper les capacités de stockage nécessaires.

L'opération est souvent réalisée en 2 temps :

- un 1^{er} passage où l'on va compter les éléments et préparer les décisions
- un 2^{d} passage où on termine le traitement

Il arrive que le comptage soit un sous-produit d'une opération précédente. Dans le chaînage vers la racine, la réduction du nombre de frères provient du choix de favoriser le plus important. C'est justement l'objet de la publication de novembre 2017 [1] que d'utiliser un compteur qui puisse mutualiser avec une opération suivante.

16.4 Ouvrir son esprit

Enfin, lorsque l'on est arrivé à un algorithme en $O(n)$ pour un traitement qui doit être exhaustif, on pourrait estimer que le contrat est rempli.

Il faut se poser malgré tout une dernière question. Peut-on dérouler des parties de l'algorithme en parallèle ?

Ce n'est pas toujours chose aisée.

Dans cette démarche, il faut également penser que la donnée que nous manipulons peut être structurée. Nous avons vu dans la section "2 TP1 : Micro-Parallélisme & gestion de cache" qu'un entier pouvait être décomposé en sous-ensemble de même taille. Par exemple, le voir comme 2 entier de y bits. Ou, si l'on préfère, l'écrire en base 2^y et considérer séparément la dizaine et l'unité.

C'est également le cas dans la section "14 TP7 : Parcours d'arbres" où l'on extrait une information d'un pointeur (si l'index est inférieur au nombre de pixels, alors le fils est une feuille, sinon c'est un nœud).

16.5 Sujet de recherche

Ce module, vous fait également prendre conscience que tout n'est pas figé dans le domaine des algorithmes.

Un IUT est une composante de l'enseignement supérieur dispensé dans une université. A ce titre, son activité est adossée à la recherche et l'adaptation au programme pédagogique national permet d'y intégrer des résultats de recherche dans les options qui vous sont proposées.

Références

[1] François Merciol, Thibaud Balem, and Sébastien Lefèvre. Efficient and large-scale land cover classification using multiscale image analysis. In *Big Data from Space*, Toulouse, France, 2017.

[2] Ronan Fablet, Nicolas Bellec, Laetitia Chapel, Chloé Friguet, René Garello, Pierre Gloaguen, Guillaume Hajduch, Sébastien Lefèvre, François Merciol, Pascal Morillon, Christine Morin, Matthieu Simonin, Romain Tavenard, Cédric Tedeschi, and Rodolphe Vadaine. Next Step for Big Data Infrastructure and Analytics for the Surveillance of the Maritime Traffic from AIS & Sentinel Satellite Data Streams. BiDS' 2017 - Conference on Big Data from Space, November 2017. Poster.

[3] Liberty Jesse, Rao Siddhartha, and Jones Bradley. *Le langage C++*. Pearson, 12 2012. Traducteurs : Olivier Engler et Nathalie Le Guillou de Penanros.

[4] Robert Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the A C M*, pages 215–225, 1975.

[5] Jiří Havel, François Merciol, and Sébastien Lefèvre. Efficient Tree Construction for Multiscale Image Representation and Processing. *Journal of Real-Time Image Processing*, 2016.

Thibaud Balem était en stage de 2^{nde} année de DUT d'informatique dans l'équipe Obelix de l'IRISA site de Vannes lors de la rédaction de l'article [1].