République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

N° d'ordre : …….
Série : …….

MEMOIRE

Présenté par

Chérif TAOUCHE

Pour l'obtention du diplôme de Magister en

Informatique

Option: Information & Computation

Thème :

Implémentation d'un Environnement Parallèle pour la Compression d'Images à l'aide des Fractales

Dirigé par: Dr M. Benmohammed

Soutenu le : … / … / 2005

Devant le jury :

Dr M. Batouche	Prof.	Université Mentouri Constantine	Président
Dr M. Benmohammed	M.C	Université Mentouri Constantine	Rapporteur
Dr M.K. Kholladi	M.C	Université Mentouri Constantine	Examinateur
Dr D. Saidouni	M.C	Université Mentouri Constantine	Examinateur

A ma mère pour son aide et support

Remerciements

Mes spéciaux remerciements vont à mon encadreur ***Mr. Benmohammed Mohamed*** *pour son aide et sa compréhension.*

Je tiens également à remercier mon meilleur amis ***Mohamed Chemachema*** *pour son aide et soutien et ses encouragements.*

Je remercie aussi mon frère ***Rabah*** *pour son aide.*

Mes sincères remerciements à ***Mme. Lehlour Rahima*** *directrice de wilaya du secteur de formation professionnelle à Mila pour son aide et son soutien.*

SOMMAIRE

Chapitre 3 La théorie Fractales : outils et fondements mathématiques

Chapitre 4 Fractale Comme Méthode De Compassion (Encodage et Décodage)

Chapitre 5 Parallélisme et architectures parallèles

Chapitre 6 Parallèlisation d'un algorithme de compression fractale

Chapitre 7 Implémentation et résultats

Introduction

La société actuelle produit un nombre croissant de données qui doivent être traitées, transmises et/ou stockées. Celles-ci sont principalement des sons, des images ou des textes et proviennent de différents secteurs tels que par exemple la physique, la médecine, la biologie, l'industrie, la culture, le tourisme ou la finance. La représentation de ces informations sous forme numérique fiabilise leur transmission au travers des réseaux informatiques et facilite leur manipulation. La numérisation présente cependant un inconvénient: elle requiert que les dispositifs de stockage ainsi que les largeurs des bandes passantes des lignes de transmission soient suffisamment importants. Ceci n'est pas toujours possible et il faut dans ce cas faire appel à des algorithmes de compression des données.

Comprimer des données revient tout simplement à en éliminer toute information superflue. On parle alors de réduction de la redondance. La redondance peut être occasionnée par une représentation non efficace des données. Son élimination, dans ce cas, n'empêche pas la restitution de ces données, et elle se fait à l'aide de méthodes de compression dites réversibles. En ce qui concerne les images, la redondance peut être psychovisuelle. En effet, les images comportent des informations non pertinentes dont l'élimination ne nuit pas au message perçu. Dans ce cas, la restitution exacte des données n'est pas possible et on parle de méthodes de compression irréversibles [1]. C'est l'une de ces méthodes irréversibles qui est au centre de notre travail qu'est la compression fractale d'images.

Cette méthode a été promue par M. Barnsley, qui a créé une société travaillant sur la compression fractale d'images, mais qui n'a pas publié les détails de son algorithme. Le premier algorithme semblable rendu publique est dû à E. Jacobs et R. Boss du Naval Ocean Systems Center de San Diego, qui utilisait une partition régulière et une classification de segments de courbes pour compresser des courbes fractales aléatoires (telles que des résultats politiques) en deux dimensions [2],[3]. Un étudiant de M. Barnsley, A. Jacquin, fut le premier à publier un algorithme semblable de compression fractale d'images [4],[5].

La méthode de compression fractale d'images est relativement récente fondée essentiellement sur la théorie des systèmes de fonction itérées d'IFS (Iterated Function System), le problème principal de cette méthode est de trouver un opérateur (un IFS particulier) qui a pour point fixe une image proche de l'image à

compresser. L'image est alors représentée par cet opérateur. La construction est obtenue en itérant l'opérateur à partir de n'importe quelle image de départ. Cette approche s'appuie sur deux théorèmes: le théorème du point fixe pour les IFS et le théorème du collage. Divers travaux ont montré que cette méthode possède un potentiel lui permettant de figurer parmi les méthodes de compression efficaces, dont les principaux avantages sont:

- Des taux de compression élevés.
- Une représentation indépendante de l'échelle: un IFS avec possibilité de zoom de l'image sans perdre de l'information.
- Une procédure de décompression particulièrement rapide et simple.

Le défaut majeur des techniques de compression fractale est leur lenteur à la phase de codage: en effet, plusieurs travaux de recherches dans ce domaine se penchés sur deux axes, l'axe des partitionnement de l'image à savoir le quadtree, HV, Delaunay et autres ainsi que l'accélération du codage avec l'élaboration des schémas hybrides et de parallèlisation d'algorithmes. Cette recherche peut être implantée naïvement, avec complexité linéaire, auquel cas la complexité globale est quadratique en taille de l'image. Mise en œuvre de cette manière, la compression d'une image de taille moyenne peut prendre quelques heures sur un ordinateur personnel [6].

Ce travail a pour thème: Implémentation d'un Environnement Parallèle pour la Compression d'Images à l'aide des Fractales.

Notre but est de prouver que la parallèlisation de l'algorithme de compression fractale permet de minimiser le temps de codage d'une manière intéressante en implémentant un environnement parallèle de compression avec cette méthode dont l'architecture utilisée est MIMD à mémoire distribuée qui est simulée par un réseau.

L'organisation générale du mémoire est décrite ci-dessous.

Le chapitre 1 est consacré à la présentation de quelques concepts sur l'image et les éléments de théorie de l'information et les caractéristiques des méthodes de compression (Rapport et taux …).

Le chapitre 2 est consacré à la présentation des principales méthodes de compression réversibles et irréversibles.

Le chapitre 3 introduit les notions nécessaires à la compréhension de la théorie des systèmes de fonctions itérées (IFS).

Le chapitre 4 présente une description de la méthode de compression fractale avec les différentes méthodes de partitionnement.

Le chapitre 5 introduit quelques notions concernant le parallélisme et les architectures parallèles.

Le chapitre 6 est consacré à la parallèlisation de l'algorithme de compression fractale de base.

Le chapitre 7 est consacré à l'implémentation parallèle de l'algorithme de compression fractale et aux résultats obtenus avec une discussion de ces derniers. Et on termine ce mémoire avec une conclusion générale.

Chapitre 1

Généra1ités et Notions de Base sur l'image

Dans ce chapitre nous allons parler de quelques concepts concernant l'image et ses différents types, afin d'étudier la compression d'image en abordant par les besoins de cette analyse.

Ensuite nous présenterons les éléments fondamentaux pour mesurer la qualité de l'image compressée.

1.1 L'image

L'image est une Représentation d'un objet par les arts graphiques ou plastiques : la sculpture, la photographie, le dessin, le film, etc. [7].
C'est aussi un ensemble structuré d'informations qui, après affichage sur l'écran, ont une signification pour l'œil humain.

Elle peut être décrite sous la forme d'une fonction F(x, y) de brillance analogique continue, définie dans un domaine borné, tel que x et y sont les coordonnées spatiales d'un point de l'image et F est une fonction d'intensité lumineuse et de couleur. Sous cet aspect, l'image est inexploitable par la machine, ce qui nécessite sa numérisation [8].

1.2 Pixel

Contraction de l'expression anglaise "Picture elements": éléments d'image, le pixel est le plus petit point de l'image, c'est une entité calculable qui peut recevoir une structure et une quantification. Si le bit est la plus petite unité d'information que peut traiter un ordinateur, le pixel est le plus petit élément que peuvent manipuler les matériels et logiciels d'affichage ou d'impression [9].

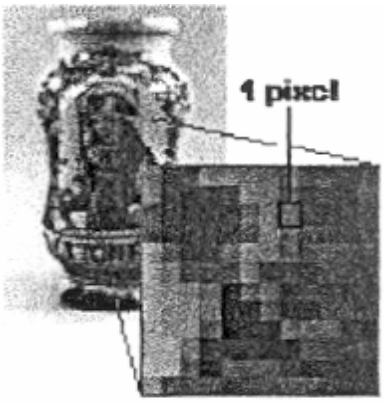

Figure 1.1 représentation de pixel

La lettre A, par exemple, peut être affichée comme un groupe de pixels dans la figure ci dessous :

Figure 1.2 L'image comme un groupe de pixels

La quantité d'information que véhicule chaque pixel donne des nuances entre images monochromes et images couleur. Dans le cas d'une image monochrome, chaque pixel est codé sur un octet, et la taille mémoire nécessaire pour afficher une telle image est directement liée à la taille de l'image.

Dans une image couleur (R. V. B), un pixel peut être représenté sur trois octets : un octet pour chacune des couleurs : rouge (R), vert (V) et bleu (B) [7].

1.3 Image numérique

Contrairement aux images obtenues à l'aide d'un appareil photo ou dessinées sur du papier, les images manipulées par un ordinateur sont numériques (représentées par une série de bits). L'image numérique est l'image dont la surface est divisée en éléments de tailles fixes appelés

cellules ou pixels, ou calculé à partir d'une description interne de la scène à représenter [8].

La numérisation d'une image peut s'effectuer selon deux procédés différents de codage, ayant chacun leurs applications propres et produisant deux modes d'images : en pixel ou vectoriel.

1.3.1 L'image en pixel (Bitmap)

L'image bitmap est représentée par une trame de points que l'on appelle pixels. Ce ne sont pas des formules mathématiques qui définissent les formes, mais un ensemble de pixels qui agissent comme un tableau pointilliste [10].

Exemple

Une image (comme un cercle par exemple) de petite taille, que l'on agrandit dix fois, Cette opération peut entraîner une perte dans la qualité des couleurs ou la netteté. Alors l'image devient déformée (figure 1.3).

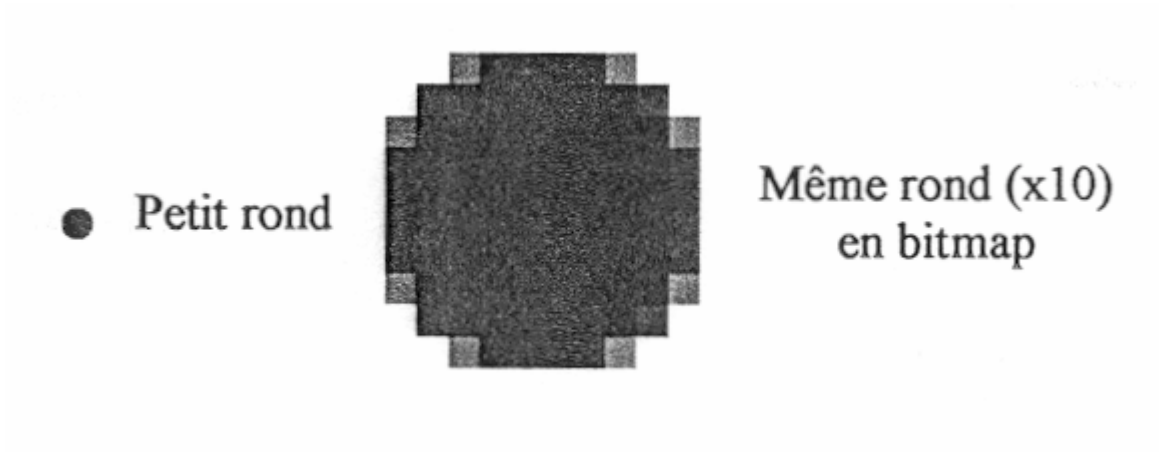

Figure 1.3

1.3.2 L'image vectorielle

L'image vectorielle est une représentation conceptuelle de forme calculée par des formules mathématiques, (exemple, un cercle n'est pas déterminé par des pixels mais par une formule mathématique qui détermine sa forme, sa taille et son emplacement) [10].

Exemple

Une image (comme un cercle par exemple) de petite taille que l'on grossit dix fois, le cercle est agrandit mais il n'est pas déformé car la précision de l'image ne dépend pas du facteur d'agrandissement (contrairement au bitmap).

Figure 1.4

1.3.3 Comparaison des images en pixels et des images vectorielles

1.3.3.1 En pixels

Applications

Dessins type artistique, image qualité photo, animations……

Logiciels

Paint, Paintbrush, Paint Shop Pro.

Extensions – format

BMP, PCX, GIF, TIF …

Avantages

- Définition de l'image au point près possibilité de dégradés infinis.

Inconvénients

- Fichiers volumineux
- Ne conserve pas les proportions lors des modifications ex. agrandissement ou réduction.
- Risque de disparition de pixels lors des modifications de taille

1.3.3.2 Vectorielle

Applications

Dessins techniques, logo, schémas, dessins de référence pour les machines à

commandes numériques (gravure, ...)[11].

Logiciels

Corel Draw, Autosketch, Autocad, Designer, Design- works, Word art...

Extensions – format

GEM, SKD, DXF, CDR, EPS, WMF...

Avantages [11]

- Fichiers de volume peu important
- Conservation des proportions lors des modifications de taille de l'image
- Dessin aux contours nets

Inconvénients

- Moindre possibilité de travail sur les couleurs
- Pas de travail sur les photos

En conclusion

- La modification de la taille d'une image vectorielle ne change pas sa qualité (netteté).
- La modification de la taille d'une image en pixels diminue sa qualité.

1.3.4 Caractéristiques d'une image numérique

L'image est un ensemble structuré d'informations caractérisées par les paramètres suivants: [8].

1.3.4.1 Dimension

C'est la taille de l'image. Cette dernière se présente sous forme de matrice dont les éléments sont des valeurs numériques représentatives des intensités lumineuses (pixels). Le nombre de lignes de cette matrice multiplié par le nombre de colonnes nous donne le nombre total de pixels dans une image.

1.3.4.2 Résolution

C'est la clarté ou la finesse de détails atteinte par un moniteur ou une imprimante dans la production d'images. Sur les moniteurs d'ordinateurs, la résolution est exprimée en nombre de pixels par unité de mesure (pouce ou centimètre).

1.3.4.3 Bruit

Un bruit (parasite) dans une image est considéré comme un phénomène de brusque variation de l'intensité d'un pixel par rapport à ses voisins, il provient de l'éclairage des dispositifs optiques et électroniques du capteur.

1.3.4.4 Histogramme

L'histogramme des niveaux de gris ou des couleurs d'une image est une fonction qui donne la fréquence d'apparition de chaque niveau de gris (couleur) dans l'image.

1.3.4.5 Contours et textures

Les contours représentent la frontière entre les objets de l'image, ou la limite entre deux pixels dont les niveaux de gris représentent une différence significative. Les textures décrivent la structure de ceux-ci. L'extraction de contour consiste à identifier dans l'image les points qui séparent deux textures différentes.

1.3.4.6 Luminance

C'est le degré de luminosité des points de l'image.

1.3.4.7 Contraste

C'est l'opposition marquée entre deux régions d'une image, plus précisément entre les régions sombres et les régions claires de cette image.

1.3.4.8 Images à niveaux de gris

Le niveau de gris est la valeur de l'intensité lumineuse en un point. La couleur du pixel peut prendre des valeurs allant du noir au blanc en passant par un nombre fini de niveaux intermédiaires [9].

Le nombre de niveaux de gris dépend du nombre de bits utilisés pour décrire la " couleur " de chaque pixel de l'image. Plus ce nombre est important, plus les niveaux possibles sont nombreux.

1.3.4.9 Images en couleurs

La représentation des couleurs s'effectue de la même manière que les images monochromes avec cependant quelques particularités. En effet, il

faut tout d'abord choisir un modèle de représentation. On peut représenter les couleurs à l'aide de leurs composantes primaires. Les systèmes émettant de la lumière (écrans d'ordinateurs,...) sont basés sur le principe de la synthèse additive : les couleurs sont composées d'un mélange de rouge, vert et bleu (modèle R.V.B.)[8], [12], [9].

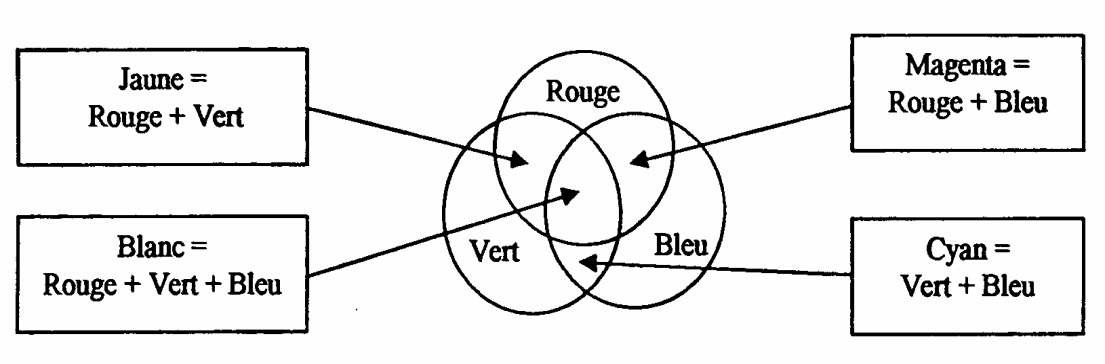

Figure 1.5 Synthèse Additive

Nombres de bits	**Nombre de couleurs**
1 bit/ pixel	2 – blanc & noir
8 bit/ pixel	256 couleurs
24 bit/ pixel	16.8 millions couleurs

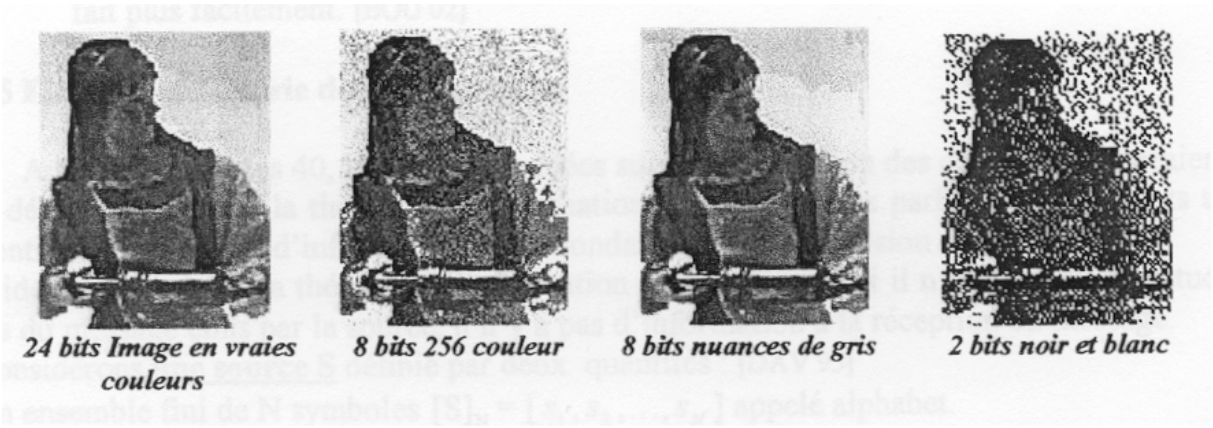

Figure 1.6 le nombre de nuances ou de couleurs dépend du nombre de bits

1.4 Avantages de la numérisation

1.4.1 Recherche [9]

- ✓ La numérisation facilite et rend plus efficace la recherche par les conservateurs, les étudiants, les enseignants, les érudits, les chargés de cours à l'université, les chercheurs et les spécialistes, car elle permet d'étudier des images disparates dans des contextes nouveaux.
- ✓ Il y a davantage d'exploration de ressources liées aux objets exposés, et l'information à propos du musée et de ses collections importantes s'en trouve améliorée.

1.4.2 Conservation

- ✓ La manipulation des originaux est réduite au minimum.
- ✓ La numérisation offre une stratégie de remplacement pour des objets, par exemple des films et des archives sonores, en voie de détérioration complète.
- ✓ La conservation des objets est mise en valeur et améliorée [9].

1.4.3 Gestion des collections

- ✓ L'information contenue dans les systèmes de gestion des collections est améliorée du fait de la présence des données de catalogage nécessaires et de la banque d'images de la collection.
- ✓ Les rapports sur l'état des objets et les rapports d'évaluation sont plus faciles à produire.
- ✓ Les fonctions de conservation sont accomplies avec plus d'efficacité.
- ✓ La gestion des déplacements d'objets est facilitée.

1.4.4 Accès par le public et promotion

- ✓ Il est possible d'illustrer et d'explorer des thèmes liés à des programmes de formation, en particulier des thèmes bien précis [9].

- ✓ La production de documents, programmes, publications, etc., à l'occasion d'expositions se fait plus facilement [13].

1.5 Eléments de théorie de l'information

A la fin des années 40, les premières idées sur la compression des données émergeaient, avec le développement de la théorie de l'information. Les chercheurs parlaient de concepts tels que l'entropie, la quantité d'information, La redondance, sous l'impulsion de C. Shannon.

L'idée principale de la théorie de l'information est de dire que s'il n y a pas d'incertitude vis a vis du message émis par la source, il n'y a pas d'information à la réception du message. Considérons une source S définie par deux quantités : [14].

Un ensemble fini de N symboles $[S]_N=[s_1,s_2,...,s_N]$ appelé alphabet.

Un mécanisme d'émission de suites de tels symboles suivant une loi de probabilité donnée $[P]_N=[p(s_1),p(s_2),...,p(s_N)]$, avec

$\sum P(s_i)=1$ et $\bigcup s_i=$ évènement certain.

1.5.1 Différents types de sources

Une source est dite ***simple*** ou ***sans mémoire*** si :

Les symboles successifs émis par la source sont indépendants (variables aléatoires indépendantes et de même loi) [14].

La probabilité d'une suite de symboles $S_N=S_{t1},S_{t2},...,S_{tn}$ émis à des instants successifs tj est donnée selon l'égalité (1). La source simple est équivalente à une suite de variables aléatoires indépendantes à valeurs dans $[S]_N$.

$$P(S_{t1},S_{t2},...,S_{tn})=P(S_{t1}),P(S_{t2}),...,P(S_{tn}) \quad \textbf{.....(1)}$$

Une source est dite de ***Markov à l'ordre r*** si l'apparition du symbole S_t est conditionnée par seulement r symboles précédents : [14]

$P(S_t|S_{t-1},S_{t-2},S_{t-3},...)=P(S_t|S_{t-1},S_{t-2},S_{t-3},...,S_{t-r})$; $\forall t$.

1.5.2 Entropie d'une source simple

L'entropie $H(S)$ d'une source simple $[S]_N$ associée à une loi de probabilité $[P]_N$ est définie selon la formule suivante : [14]

$$H(S)=-\sum_{i=1}^{N}P(S_i)\log_2(P(S_i)) \text{ bits}$$

$H(S)$ est une appréciation numérique globale attachée à une réalisation de la source. Elle définit l'information moyenne de chaque message (symbole) de la source.

$H(S)$ est maximale si tous les symboles $[S_1,S_2,...,S_N]$ de $[S]_N$ sont équiprobables.

On a alors : $P(S_i)=\frac{1}{N}$ et $H(S)=\log_2 N$ bits .

Dans ce cas l'entropie de la source est égale à l'information associée à chaque message pris individuellement.

Exemple

Au jeu de pile ou face (source binaire) on associe deux messages Equiprobables. La probabilité $P(S_i)$ est égale à $\frac{1}{2}$ et l' entropie $H(S)$ à 1 bit.

1.6 Traitements généraux

Les traitements généralement appliqués aux images sont 2 types :

1.6.1 Amélioration ou retouche de l'image

Le traitement de l'image informatique correspond à toutes les possibilités de travail, retouches ou modifications que l'on peut réaliser avec un ordinateur sur une image (enregistrée) quand elle est affichée à l'écran.

Les logiciels de traitement d'image en pixels permettent de réaliser des modifications sur ces images pour obtenir des effets précis : [11]

- recadrer une image
- modifier la taille (à l'écran ou à l'impression)
- retoucher des couleurs
- intégrer un texte
- changer un fond
- effacer une partie de l'image...

1.6.2 La Compression

La qualité d'une image dépend de sa résolution (nombre de pixels) et du nombre de couleurs utilisées pour coder chaque pixel.

- En général IMAGE =GROS FICHIER.
- Le volume d'une image numérisée est égal à la quantité d'octets utilisés pour la coder.

Exemples

800 x 600 pixels représentent 480 000 points à mémoriser

C'est-à-dire 480 000 octets pour une image en noir et blanc avec 256 nuances de gris

et 480 000 x 3 = 1,4 MO pour une image en couleur avec nuancier de 16 millions de couleurs. Pour diminuer le volume des fichiers, on peut soit:

- Modifier le nombre de couleurs
- Sauvegarder dans un format dit de compression.

Conséquences :

Lors du travail sur l'image, il ne faut pas faire de sauvegardes intermédiaires en format compressé car il y a alors une perte de la définition de l'image à chaque enregistrement. Attention avec certains logiciels (PSP), les résultats de la compression ne sont pas toujours visibles à l'écran (conservation de l'image de départ) mais seulement au redémarrage du logiciel ou lorsque l'image est réouverte.

1.6.2.1 Le but de la compression d'image

Les images numériques sont des fichiers volumineux qui occupent beaucoup d'espace sur disque ou allongent considérablement les temps de transmission sur réseau.

La compression d'images permet de réduire énormément la taille des images. Par exemple, pour une image 640 x 480 pixels en vraies couleurs :

- Une seule image non compressée peut être enregistrée sur une disquette 1,4 Mo. Avec la compression d'images, une disquette peut contenir entre 20 et 50 images.

- Le temps de lecture non compressée est de 3-seccondes sur un lecteur de cd-rom 8x. Avec compression d'image, cette durée peut être réduite à environ 1 seconde, y compris le temps de décompression.
- Le temps de transmission non compressée est de 3 minutes via un modem 28 Kbps. Après compression, la durée de transmission est réduite à une dizaine de secondes.

1.6.2.2 Caractéristiques des méthodes de compression

1.6.2.2.1 Rapport et taux de compression

Le rapport de compression est l'une des caractéristiques les plus importantes de toutes les méthodes de compression, il représente le rapport entre le nombre de bits de la forme canonique au nombre de bits après codage : [15], [16]

$$C_R = rapport_c = \frac{nombre\ bits\ avant\ compression}{nombre\ bits\ aprés\ compression}$$

le taux de compression est un pourcentage de l'espace obtenu après la compression par rapport à l'espace total requis par les données avant la compression

$$T_c = taux_c = (1 - \frac{1}{rapport\ de\ compression}) * 100\ .$$

Cela indique :
Qu'un fichier compressé indique à sa taille originale aura un taux de compression de 0 %. Un fichier réduit à 0 octet, aura un taux de compression de 100%.

1.6.2.2.2 Mesure de distorsion

Pour mesurer la distorsion entre l'image reconstruite et l'image originale (Mesure de la qualité visuelle de l'image reconstruite) on va utiliser l'Erreur

Quadratique Moyenne MSE (Mean Square Error) ou du rapport signal à bruit PSNR (Peak Signal to Noise Ratio). [16]

Etant donnée une image originale composée de pixels $a_i(i=1...N)$ et l'image décodée composée de pixels $\hat{a}_i(i=1...N)$.

Alors l'erreur quadratique moyenne est donnée par :

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(a_i - \hat{a}_i)^2$$

Le **PSNR** est donnée par :

$$PSNR = 10\log_{10}\frac{(2^R-1)^2}{MSE} \qquad DB(décibels).$$

En compression d'images le PSNR d'une image de taille 512 par 512 pixels, chaque pixel est codé sur 8 bits est défini par :

$$PSNR = 10\log_{10}\frac{(255)^2}{MSE}$$

1.6.2.3 Nature des images

Les images à compresser peuvent être de différentes natures : [14]

- **images photographiques :** ce sont généralement des images de scènes naturelles dans lesquelles l'intensité lumineuse varie de manière relativement continue.
- **les images** « **modales** » dans lesquelles l'intensité lumineuse est très changeante localement (dessins manuels par exemple). Leur histogramme est multimodal.

La numérisation des images peut aussi se faire de différentes manières, retournant : [14]

- **les images binaires** qui contiennent seulement deux niveaux de *gris* différents.
- **les images multi -niveaux** numérisées sur plus d'un (1) bit par pixel.

- **les images multi- canaux** dont l'exemple classique est celui des images couleurs.

1.7 Conclusion

La représentation des images fixes est l'un des éléments essentiels des applications multimédias, comme dans la plupart des systèmes de communication.

La manipulation des images pose cependant des problèmes beaucoup plus complexes que celles du texte. En effet, l'image est un objet à deux dimensions, censé représenter un espace à trois dimensions, ce qui a deux conséquences majeures : [7]

- Le volume des données à traiter est beaucoup plus important.
- La structure de ces données est nettement plus complexe.

Il en résulte que la manipulation, le stockage et la représentation de ces données se heurtent à certaines limitations.

Grâce au traitement d'image, ces contraintes sont levées ou contournées. En effet, ce domaine cherche à détecter la présence de certaines formes, certains contours ou certaines textures d'un modèle connu, c'est le problème de la détection [8].

Un autre aspect de traitement d'image concerne l'analyse et la compréhension de l'image dans le but d'en extraire des informations utiles. Il cherche aussi parfois à comprimer l'image, afin de gagner en vitesse lors de la transmission de l'information, et en capacité de stockage, tout en dégradant le moins possible les images considérées [8].

C'est le domaine du codage ou compression des images dont notre travail fait partie. Le chapitre suivant est consacré aux différentes méthodes de compression.

Chapitre 2

Les Différentes Méthodes de Compression

Après avoir présenter les différentes notions de base concernant la compression d'image nous étudions dans ce qui suit les différents types de compression de cette dernière et nous détaillons ces méthodes l'une après l'autre.

2.1 Compression physique et logique

Une distinction doit cependant être faite entre la compression physique et la compression logique [17], [18]:

- **La compression physique :** s'applique uniquement aux données de l'image. Il s'agit de translater les trains de bit d'un motif à un autre.
- **La compression logique :** est effectuée par un raisonnement logique en substituant une information par une information équivalente.

2.2 Compression symétrique et asymétrique

Les algorithmes de compression peuvent être divisés en deux groupes : les algorithmes symétriques et les algorithmes asymétriques [10].

Dans le cas de ***la compression symétrique,*** la même méthode est utilisée pour compresser et décompresser l'information, il faut donc la même quantité de travail pour chacune de ces opérations. C'est ce type de compression qui est utilisée dans les transmissions de données.

La compression asymétrique demande plus de travail pour l'une des deux opérations; si par exemple, on désire stocker une image dans une base de données, il est intéressant de la stocker de manière à ce quelle tienne un minimum de place sur le disque, et qu'elle soit le plus rapidement possible décompressée. Les algorithmes nécessitant plus de travail à la décompression ne sont pas courants.

2.3 Les différentes méthodes de compression

Les méthodes de compression d'images sont classées en deux catégories [17],[18].

La compressions ***sans pertes (réversible)*** restituent l'image originale après un cycle de compression / décompression. La compression sans pertes est très efficace avec les images 1 à 8 bits/pixel du type cartes, dessins et plans. Cette compression a une efficacité réduite pour les photographies 24 bits/pixel ou plus. Elle est utilisée pour les formats GIF, TIFF, PNG.

Généralement, la compression réversible regroupe les techniques pour générer une copie exacte des données après un cycle de compression / décompression. Le stockage des enregistrements d'une base de données, des tableurs ou des fichiers de traitement de texte utilisent ce type de compression. Dans ces applications, la perte d'un seul bit peut être catastrophique.

La compression avec pertes (irréversible) élimine certaines informations qui ne sont pas indispensables ; pour l'appréciation visuelle des images par un oeil humain, ainsi qu'avoir le meilleur taux de compression possible. Les pertes dues à la compression se traduisent par des artefacts tels que du flou sur les transitions ou des distorsions géométriques ou des réductions des couleurs. La compression irréversible montre toute son efficacité lorsqu'elle est appliquée aux images numérisées. Cette compression est utilisée pour les formats JPEG par exemple [10].

2.3.1 La compression réversible

Pour ce type de compression nous présentons dans ce qui suit les algorithmes les plus connus et utilisés, citant:

2.3.1.1 L'algorithme de Shannon-Fano

Il utilise des codes de longueur variable, comportant d'autant plus de bits que la probabilité du symbole est faible. Les codes sont définis à l'aide d'un algorithme spécifique selon un arbre de Shannon-Fano:

- Les symboles sont triés et classés en *fonction* de leur fréquence en commençant par le plus fréquent.
- La liste des symboles est ensuite divisée en deux parties de manière à ce que le total des fréquences de chaque partie soit aussi proche que possible.
- Le chiffre binaire 0 est affecté à la première partie de la liste, le chiffre 1 à la deuxième partie.
- Chacune des deux parties fait à son tour l'objet des démarches 2 et 3.

Et ainsi de suite jusqu'à ce que chaque symbole soit devenu une feuille de l'arbre correspondant à un code déterminé.

Exemple

Soit la table des fréquences (ou probabilités) des symboles :

Symbole	**Fréquence**
A	*7*
B	6
C	5
D	14
E	4

Effectuons un classement hiérarchique et une première division :

D	14	22	0	
A	7		0	**1 ère division**
B	6	16	1	
C	5		1	
E	4		1	

D	14		0	0		**2 ème division**
A	7		0	1		
B	6		1	1		
C	5		1	0	0	**3 ème division**
E	4		1	0	1	**4 ème division**

L'ordre de Shannon-Fano correspondant est simple :

	0			1	
0	1			0	1
					B
D	A		0	1	
			C	E	

Les codes attachés aux divers symboles sont:

D	0	0	
A	0	1	
B	1	1	
C	1	0	0
E	1	0	1

Le nombre de bits est plus faible lorsque la fréquence est importante. Sauf dans des cas extrêmement simples, cette méthode ne permet pas d'approcher efficacement l'entropie. On lui préfère l'algorithme de Huffman [17].

2.3.1.2 Le codage de Huffman

C'est David Huffman qui l'a mis au point en 1952. Ce type de compression donne de bons taux de compressions, notamment pour les images monochromes (les fax par exemple).

Le But : réduire le nombre de bits utilisés pour le codage des caractères fréquents dans un texte et d'augmenter ce nombre pour des caractères plus rares [18].

Algorithme de compression

- On cherche la fréquence des caractères.
- On trie les caractères par ordre décroissant de fréquence.
- On construit un arbre pour donner le code binaire de chaque caractère.

(Voir les détailles de l'algorithme dans l'Annexe A)

Construction de l'arbre : on relie deux à deux les caractères de fréquences les plus basses et on affecte à ce nœud la somme des fréquences des caractères. Puis on répète ceci jusqu'à ce que l'arbre relie toutes les lettres. L'arbre étant construit, on met un 1 sur la branche à droite du nœud et un 0 sur celle de gauche [18].

Exemple

La démarche peut être appliquée aux symboles et fréquences de l'exemple précédent.

Fréquences	14	7	6	5	4
Symboles	D	A	B	C	E

1ère phase

			0	1
			9	
14	7	6	5	4
D	A	B	C	E

2ème phase

		0	1			
		13				1
14		7	6		9	
D		A	B		C	E

3ème phase

			0		1	
				22		
		0	1		0	1
14		13			9	
D		A	B		C	E

4ème phase

1				0		
			0		1	
				22		
		0	1		0	1
14		13			9	
D		A	B		C	E

D'ou les codes des divers symboles :

D	1		
A	0	0	0
B	0	0	1
C	0	1	0
E	0	1	1

On vérifie que les codes les plus courts correspondent aux symboles les plus fréquents.

On remarque que pour le même symbole et la même fréquence, les codes de Shannon et de Huffman ont souvent des longueurs différentes.

Si l'on compare dans cet exemple, le nombre total de bits nécessaires pour coder les symboles en tenant compte de leur fréquence, on a :

Symbole	**Fréquence**	**Nombre de bits Codage Shannon**	**Nombre de bits Codage Huffman**
D	14	14 x 2=28	14 x 1=14
A	7	7 x 2=14	7 x 3=21
B	6	6 x 2=12	6 x 3=18
C	5	5 x 3=15	5 x 3=15
E	4	4 x 3=12	4 x 3=12
TOTAL		91	80

On constate, et c'est une propriété générale, que le codage de Huffman est le plus performant.

2.3.1.3 Le codage arithmétique

Il a été prouvé que le codage de Huffman est la meilleure méthode de codage à codes de longueur fixée. Mais les codes de Huffman doivent avoir une longueur entière de bits, ce qui ne permet pas toujours de réaliser une compression optimale. Le codage dit arithmétique est alors une méthode plus performante [18].

Procédure de codage arithmétique [14]

- Calculer la probabilité associée a chaque symbole dans la chaîne à coder.
- Associer à chaque symbole un sous intervalle proportionnel à sa probabilité, dans l'intervalle [0.1] (l'ordre de rangement des intervalles sera mémorisé car il est nécessaire au décodeur)
- Initialiser la limite inférieure de l'intervalle de travail à la valeur 0 et la limite supérieure à la valeur 1.

  Tant qu'il reste un symbole dans la chaîne à coder:
  - largeur = limite supérieure - limite inférieure.
  - limite inférieure =limite inférieure + largeur * (limite basse du sous intervalle du symbole).
  - limite supérieure = limite inférieure + largeur * (limite haute du sous intervalle du symbole).
- La limite inférieure code la chaîne de manière unique.

On remarque que le premier symbole de la chaîne fixe est le premier chiffre après la virgule du code final.

Décrivons le principe du codage et de décodage sur un exemple :

Considérons le codage du message ATLAS. Les probabilités des caractères sont les suivantes :

Caractère	Probabilité
A	1/10
L	1/5
S	1/10
T	1/10

Dans l'intervalle général de probabilité [0,1], chaque symbole se voit affecter un intervalle de probabilité (la manière d'affecter cet intervalle n'ayant pas d'importance).

Caractère	Probabilité	Intervalle
A	1/10	$0{,}10 \leq r < 0{,}20$
L	1/5	$0{,}20 \leq r < 0{,}40$
S	1/10	$0{,}40 \leq r < 0{,}50$
T	1/10	$0{,}50 \leq r < 0{,}60$

Le codage s'effectue progressivement à partir de la première lettre du message. Puisque c'est A, le codage du message doit être compris entre 0,10 et 0,20. La deuxième lettre est T; dans le sous intervalle 0,10 à 0,20, le codage sera compris entre 50 et 60 %. Autrement dit, à ce stade (AT) sera compris entre 0,10+(0,20-0,10)x0,50 et 0,10+(0,20-0,10)x0,60 , soit entre 0,15 et 0,16.

Au niveau de la troisième lettre L, il doit être compris entre 0,15+(0,16-0,15)x0,20 et 0,15+(0,16-0,15)x0,40 soit entre 0,152 et 0,154 .

En continuant, on arrive au tableau suivant :

Caractère successif	Limite inférieure	Limite supérieure
A	0,1	0,2
T	0,15	0,16
L	0,152	0,154
A	0,1522	0,1524
S	0,15228	0,1523

Le codage du message est constitué par la dernière limite inférieure, 0,15228.

Le décodage s'effectuera de manière unique à partir de ce nombre, suivant le mécanisme inverse. Puisque 0,15228 est compris entre 0,1 et 0,2 la première lettre du message est *A*. On peut maintenant soustraire la limite inférieure

correspondant à A, soit 0,10 ce qui donne 0,05228 et diviser ce résultat par la longueur de l'intervalle correspondant à A, soit 0,10 ce qui donne : 0,5228 . Ce nombre étant compris entre 0,5 et 0,6 , la deuxième lettre est T. En continuant, on trouve dans l'ordre tous les symboles du message.

Nombre codé	Limite inférieure	Limite supérieure	Symbole
0,15228	0,1	0,2	A
0,5228	0,5	0,6	T
0,228	0,2	0,4	L
0,14	0,1	0,2	A
0,4	0,4	0,5	S

Cette technique se montre un peu plus lente que celle de Huffman mais elle présente des taux de compression supérieurs [9].

2.3.1.4 RLE (Run Length Encoding)

But : cet algorithme élide les répétitions successives de caractères.

Algorithme de compression [18]

1. Recherche des caractères répétés plus de n fois (n fixé par l'utilisateur)
2. Remplacement de l'itération de caractères par :
   - un caractère spécial identifiant une compression.
   - le nombre de fois où le caractère est répété.
   - le caractère répété.

(L'algorithme est plus détaillé dans l'Annexe A)

Algorithme de décompression

Durant la lecture du fichier compressé, lorsque le caractère spécial est reconnu, on effectue l'opération inverse de la compression tout en supprimant ce caractère spécial [18].

Exemple :

AAAAARRRRRROLLLLBBBBBUUTTTTTT

On choisit comme caractère spécial : @ et comme seuil de répétition : 3

Après compression : @5A@6RO@4L@5BUU@6T gain : 11 caractères soit 38%.

Utilité : essentiellement pour la compression des images (car une image est composée de répétitions de pixels, de couleur identique, codés chacun par un caractère).

Caractéristiques de compression [19]

- algorithme très simple.
- taux de compression relativement faible (40%).

C'est une méthode utilisée par de nombreux formats d'images (BMP, PCX, TIF).

2.3.1.5 Les algorithmes LZ**

C'est en 1977 que Abraham Lempel et Jacob Ziv ont créé le compresseur LZ77, qui est à la base de tous les algorithmes à dictionnaire que nous utilisons actuellement [17], il était alors utilisé pour l'archivage (les formats ZIP, ARJ et LHA l'utilisent).

En 1978 ils créent le compresseur LZ78 spécialisé dans la compression d'images (ou tout type de fichier de type binaire).

En 1984, Terry Welch le modifia pour l'utiliser dans des contrôleurs de disques durs, son initiale vint donc se rajouter à l'abréviation LZ pour donner LZW.

2.3.1.5.1 La méthode LZW (Lempel Ziv Welch)

La méthode LZW a certaines caractéristiques : [18], [10], [12]

- Elle est plus performante que les méthodes statistiques.
- Elle est utilisée notamment dans les formats TIFF et GIF.
- Cette méthode est peu efficace pour les images mais donne de bons résultats pour les textes et les données informatiques en général (plus de 50%).
- Elle utilise un dictionnaire qu'elle construit dynamiquement, au cours de la compression et de la décompression, qui n'est pas stocké dans le fichier compressé.
- Elle comprime en une seule lecture.
- Elle a besoin d'un apprentissage pour être efficace, et reconnaître des longues chaînes répétées. Elle est donc peu performante sur les petits fichiers.

(Voir les détailles de l'algorithme dans l'Annexe A)

La compression consiste à éviter les répétitions, pour économiser de la place.

Un dictionnaire contenant toutes les répétitions est crée lors des deux opérations. Il doit être construit de la même manière, à la compression et à la décompression, et contenir les mêmes informations [19].

Tous les ensembles de lettres qui sont lus sont placés dans le dictionnaire et sont numérotés. A chaque fois qu'un ensemble est lu, on regarde s'il en existe déjà un qui est identique. Si c'est le cas, on émet son numéro vers le fichier compressé. Sinon, on le rajoute à la fin du dictionnaire, et on écrit chacune des lettres dans le fichier compressé.

Quand on écrit un numéro au lieu d'écrire des lettres, il y a un gain de place. Mais pour cela, il faut déjà avoir beaucoup de chaînes dans le dictionnaire. L'apprentissage est donc nécessaire pour que la méthode soit efficace.

Exemple

Phrase à encoder DAD DADA DADDY DADO

Code Lu	**Code émit**	**Index dans le dictionnaire**	**Nouvelle chaîne du dictionnaire**
‘ D ‘	/	/	/
‘ A ‘	‘ D ‘	256	‘ DA ‘
‘ D ‘	‘ A ‘	257	‘AD ‘
‘ ‘	‘ D ‘	258	‘ D ‘
‘ A ‘	/	/	/
‘ D ‘	256	260	‘ DAD ‘
‘ A ‘	/	/	/
‘ ‘	256	261	‘ DA ‘
‘ D ‘	/	/	/
‘ A ‘	259	262	‘ DA ‘
‘ D ‘	/	/	/
‘ D ‘	257	263	‘ADD ‘
‘ Y ‘	‘ D ‘	264	‘ DY ‘
‘ ‘	‘ Y ‘	265	‘ Y ‘
‘ D ‘	/	/	/
‘ O ‘	‘ D ‘	267	‘ DO ‘
/	‘ O ‘	/	/

Le compresseur lit le symbole D. Il est déjà dans le dictionnaire. On passe au symbole suivant. On lit le symbole 'A', puis on l'ajoute à D ce qui devient la chaîne 'DA'. Or celle-ci n'est pas dans le dictionnaire. Le code de 'D' est émis,

puis on l'ajoute à la chaîne 'DA'. On recommence avec 'A' comme caractère initial. On suit le même processus avec 'D', ' ', 'D', en créant les entrées 'AD', 'D ', ' D' dans le dictionnaire. Il se trouve avec 'D' comme caractère initial. Il lit 'A' et forme 'DA' qui existe dans le dictionnaire, il passe donc au caractère suivant qui est 'D'. La chaîne résultante n'étant pas dans le dictionnaire, il le code 'DA' qui est 256 puis continue avec 'D' comme caractère initial. Et ainsi de suite jusqu'à la lecture de 'O' puis l'émission finale de 'O'. La chaîne est «DAD<256><256><256><259><257>DY<262>DO».

La Décompression

« DAD<256><256><259><257>DY<262>DO...»

Ancien caractère	**Nouveau caractère**	**Chaîne émise**	**Index du dictionnaire**	**Chaîne du dictionnaire**
‘ ‘	‘D‘	‘D‘	/	/
‘D‘	‘A‘	‘A‘	256	‘DA‘
‘A‘	‘D‘	‘D‘	257	‘AD‘
‘D‘	‘ ‘	‘ ‘	258	‘D ‘
‘ ‘	256	‘ DA‘	259	‘ D‘
256	256	‘DA‘	260	‘DAD‘
256	259	‘ D‘	261	‘DA‘
259	257	‘AD‘	262	‘ DA‘
257	‘D‘	‘D‘	263	‘ADD‘
‘D‘	‘Y‘	‘Y‘	264	‘DY‘
‘Y‘	262	‘DA‘	265	‘Y ‘
262	‘D‘	‘D‘	266	‘ DAD‘
‘D‘	‘O‘	‘O‘	267	‘DO‘

« DAD DADA DADDY DADO...»

Le compresseur lit le premier symbole ici 'D', puis l'émet tout simplement. Il lit le symbole 'A' puis, le compresseur l'émet, construit la chaîne 'DA' composée du dernier symbole lu et du premier caractère du symbole courant, soit 'A' car il n'y a qu'un seul caractère dans le symbole courant, et l'ajoute au dictionnaire. De même pour 'D' et ' ' ce qui rajoute les entrées 257 et 258 du dictionnaire. Il l'émet sur le flux de sortie et compose encore la chaîne 'D' (le

dernier caractère ' ' et le premier caractère ' ' et le premier caractère de la chaîne courante 'D') qu'il ajoute au dictionnaire. Et ainsi de suite.

2.3.2 La compression irréversible

Pour ce type de compression nous présentons dans ce qui suit les algorithmes les plus connus et utilisé, citant:

2.3.2.1 Quantification Scalaire (QS)

Un quantificateur scalaire est un opérateur qui associe à une variable continue u une variable discrète u' pouvant prendre un nombre plus faible, et fini de valeurs.

Il est à noter que les méthodes de codage utilisant un quantificateur ne sont jamais réversibles parce que l'étape de quantification introduit inévitablement une distorsion.

Pour un nombre de niveaux de quantification fixé L, on peut choisir les régions u' de décision $\{t_k, k=1...L+1\}$ ainsi que les seuils de décision $\{r_1...r_L\}$ de façon à minimiser la distorsion entre l'entrée et la sortie. Si u se trouve dans la région $\{t_k, t_{k+1}\}$, u' aura pour valeur r_k [14].

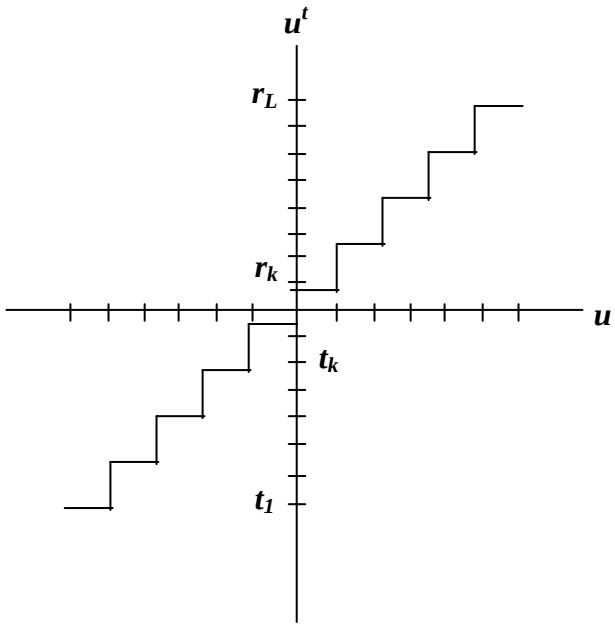

Figure 2.1 Quantification scalaire uniforme

2.3.2.2 Quantification Vectorielle (QV)

Les techniques de compression d'images exploitent généralement la redondance statistique présente dans l'image. Différentes méthodes effectuent une quantification scalaire sur des échantillons (pixels individuels de l'image).

La quantification vectorielle, développée par Gersho et Gray (1980) fait aujourd'hui l'objet de nombreuses publications dans le domaine de la compression des images numériques.

Principe de la quantification vectorielle

Un quantificateur peut être vu comme une application Q associant à chaque vecteur d'entrée $X_i=(x_i;i=1...K)$ un vecteur $Y_i=(y_i;i=1...K)=Q(X_i)$ [20].

Choisi parmi un dictionnaire de taille finie, $C=(\hat{X}_i;i=1...N_c)$. C = $(X_i \ ; \ i = 1 \ ... \ N_c)$. C peut être vu comme un catalogue de formes.

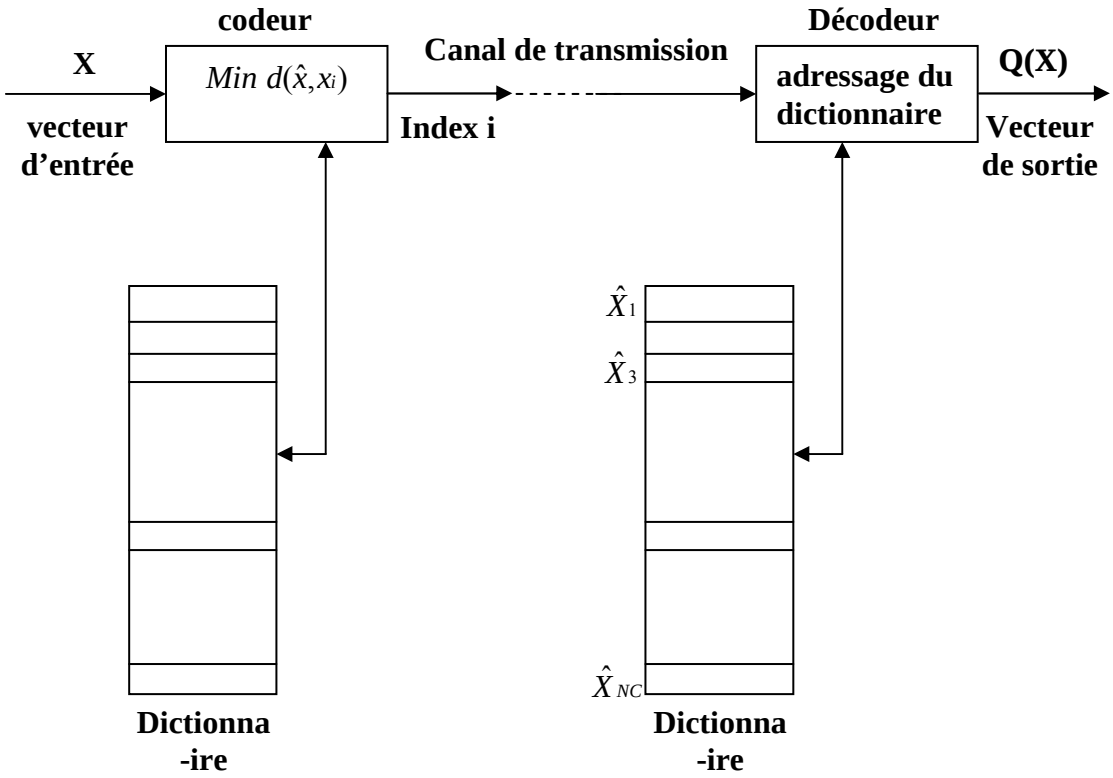

Figure 2.2 Principe du Quantificateur vectoriel

Le quantificateur est complètement décrit par :

- le dictionnaire (codebook) C .

- le partitionnement $S=(s_i;i=1...Nc)$ qui divise l'espace d'entrée en N_c vecteurs X_i, et qui leur fait correspondre un vecteur (reconstruit) $Q(X_i)=\hat{X}_i$.

La quantification vectorielle peut aussi être vue comme une combinaison de deux fonctions:

- un codeur qui prend en entrée un vecteur X_i et qui recherche dans un dictionnaire l'adresse du vecteur lui ressemblant le plus.
- un décodeur qui reçoit l'adresse et génère le vecteur $\hat{X}_i$ correspondant du dictionnaire. Ce vecteur est une approximation du vecteur X_i à coder [14].

Calcul du taux de compression

Le débit est donné par $R=\frac{\log_2 N_c}{k}$ bits par pixel .Il est lié à la dimension k des vecteurs à coder de la séquence d'apprentissage et à la taille du dictionnaire

Exemple

Pour un débit fixé à un bit par pixel et des vecteurs composés de blocs d'image de taille 2 x 2, le dictionnaire doit contenir 16 vecteurs (16 vecteurs représentant $256^{'}$ vecteurs possibles). Si la taille des vecteurs de l'image à coder est de 4x4 , le dictionnaire contient 2^{16} vecteurs (représentant 256^{16} vecteurs possibles) [14], [21].

2.3.2.3 La DCT (Transformée en Cosinus Discret)

Cette technique est une variété de la transformée de Fourier, proposée par le professeur (RAO) de l'université de Texas en 1974. Elle prend un ensemble de points d'un domaine spatiale et les transforme en une représentation identique dans un domaine de fréquences (spectrale).

Il existe une fonction DCT inverse (IDCT 'Inverse DCT') qui peut convertir la représentation spectrale d'un signal en sa représentation spatiale.

Suivant, sa simplicité et son algorithme de calcul rapide la transformation discrète en cosinus s'est imposée comme la transformation de codage de l'image [15].

L'image est découpée en sous-images ou blocs de taille fixe (8 x 8, 16 x 16, …) (Matrice carrée N x N) qui sont ensuite transformés par la transformation en cosinus (DCT).

La matrice obtenue est une matrice de même taille que le bloc de départ et les coefficients s'écrivent en fonction des coefficients du plan d'image :

$$DCT(i,j)=1/\sqrt{2n}C(i)C(j)\sum_{x=0}^{n-1}\sum_{y=0}^{n-1}pixel(x,y)Cos[(2x+1)i\pi/2n]Cos[(2y+1)j\pi/2n]$$

$$C(x)=\begin{cases}1/\sqrt{2} & si\ x=0\\ 1 & si\ x>0\end{cases}$$ DCT.

$$pixel(x,yj)=1/\sqrt{2n}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}C(i)C(j)DCT(i,j)Cos[(2x+1)i\pi/2n]Cos[(2y+1)j\pi/2n]$$

$$C(x)=\begin{cases}1/\sqrt{2} & si\ x=0\\ 1 & si\ x>0\end{cases}$$ DCT inverse.

2.3.2.4 La compression JPEG

Une des meilleures compressions actuellement sur le marché est la compression JPEG (JPEG signifiant Joint Photographie Experts Group).

En 1982, l'ISO format un groupe d'expert de la photographie (PEG) afin de réfléchir sur les méthodes permettant de transmettre des images fixes, des animations et du texte sur le réseau RNIS.

En 1986, un groupe de l'IUT-T débuta une étude sur les méthodes de compression de documents fax en niveau de gris ou en couleur. La méthode de compression élaborée était très proche de celle mise au point par PEG.

Ainsi, en 1987, l'ISO et RUT-T unirent leurs forces et formèrent un comité conjoint d'expert de la photographie (JPEG) [17].

2.3.2.4.1 Principe de compression du JPEG

La norme JPEG de 91 décrit le format des données compressées et le schéma de codage et de décodage. Les algorithmes de compression sont proposés mais n'ont pas de valeur normative [17].

Voyons d'abord le schéma de principe

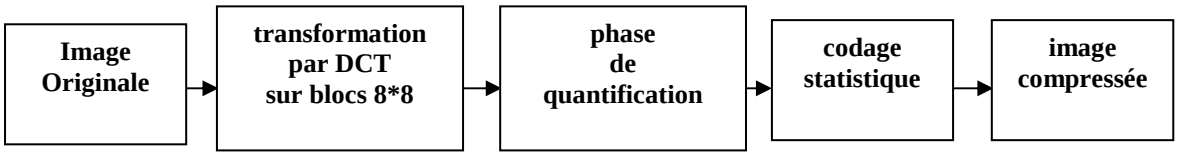

Figure 2.3 Les étapes de la Compression

Pour retrouver l'image décompressée, il faut faire le chemin inverse, mais on ne pourra pas avoir la même image [10].Voir le schéma de principe en dessous:

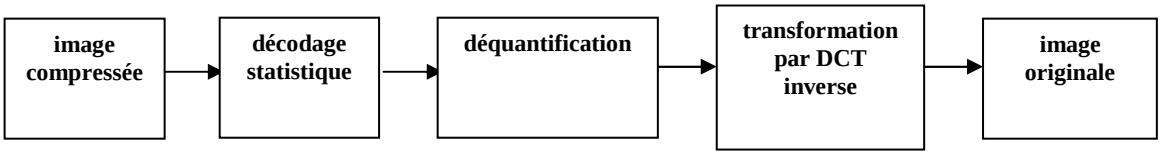

Figure 2.4 Les étapes de la décompression

Exemple

Soit un bloc de *8 x 8* pixels à *256* niveaux de gris:

Matrice de pixels d'entrée

140	144	147	140	140	155	179	175
144	152	140	147	140	148	167	179
152	155	136	167	163	162	152	172
168	145	156	160	152	155	136	160
162	148	156	148	140	136	147	162
147	167	140	155	155	140	136	162
136	156	123	167	162	144	140	147
148	155	136	155	152	147	147	136

Ensuite la Transformée en Cosinus Discrète (DCT) est appliqué sur les pixels de chaque bloc.

Matrice DCT

1210	-18	15	-9	23	-9	-14	-19
21	-34	26	-9	-11	11	14	7
-10	-24	-2	6	-18	3	-20	-1
-8	-5	14	-15	-8	-3	-3	8
-3	10	8	1	-11	18	18	15
4	-2	-18	8	8	-4	1	-7
9	1	-3	4	-1	-7	-1	-2
0	-8	-2	2	1	4	-6	0

Les valeurs de la matrice DCT ont été arrondies à l'entier le plus proche. Le coefficient continu 1210 sur la composante (0,0) représente "la moyenne" de la grandeur d'ensemble de la matrice d'entrée. Cette moyenne n'est pas une moyenne statique mais un nombre proportionnel à la somme de toutes les valeurs du signal. Les autres valeurs de la DCT sont les "écarts" par rapport à cette moyenne. Lorsque l'on monte dans les hautes fréquences, les valeurs de la matrice ont tendance à s'approcher de 0 quand on s'éloigne du coin supérieur gauche de la matrice.

L'étape de la quantification de chaque bloc regroupe les ensembles de valeurs proches. Ensuite, chaque amplitude originale sera remplacée par la valeur moyenne de l'intervalle, c'est à dire l'étape de quantification est de diminuer la précision du stockage des entiers de la matrice DCT pour diminuer le nombre de bits occupés par chaque entier. C'est la partie non conservative de la méthode. Les basses fréquences sont conservées, la précision des hautes fréquences est diminuée. La perte de précision est plus grande lorsqu'on s'éloigne de la position (0,0). Les valeurs de la matrice DCT seront divisées par la matrice de quantification.

Matrice de quantification

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Matrice DCT quantifiée

403	-4	2	-1	2	-1	-1	1
4	-5	3	-1	-1	1	1	0
-1	-3	0	0	-1	0	-1	0
-1	0	1	-1	0	0	0	1
0	1	1	0	-1	1	1	0
0	0	-1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Lors de la décompression, il suffira de multiplier la valeur de la matrice DCT quantifiée par l'élément correspondant de la matrice de quantification pour obtenir une approximation de la DCT. La matrice obtenue est appelée matrice DCT déquantifiée.

Matrice DCT déquantifiée

1209	-20	14	-9	22	-13	-15	-17
20	-35	27	-11	-13	15	17	0
-7	-27	0	0	-15	0	-19	0
-9	0	13	-15	0	0	0	0
0	13	15	0	-19	21	23	25
0	0	-17	0	0	0	0	0
15	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Matrice de pixels de sortie (décompression)

142	143	154	141	133	153	179	179
139	152	129	151	144	154	163	181
150	156	139	166	162	163	154	172
163	145	160	153	151	153	145	154
168	150	156	145	140	139	141	159
148	164	133	164	158	140	136	163
130	159	123	164	165	140	134	145
148	156	140	148	159	146	153	141

Le codage de la matrice DCT quantifiée se fait en parcourant les éléments dans l'ordre imposé par une séquence appelée Séquence zigzag. Les éléments sont parcourus en commençant par les basses fréquences puis ensuite en traitant les fréquences de plus en plus élevées.

Etant donné qu'il y a beaucoup de composantes de hautes fréquences qui sont nulles dans la matrice DCT, la séquence zigzag engendre de longues suites de 0 consécutifs. D'une part, les suites de valeurs nulles sont simplement codées en donnant le nombre de 0 successifs.

D'autre part, les valeurs non nulles seront codées en utilisant une méthode statique.

2.3.2.4.2 Utilité et caractéristiques de la compression JPEG [17], [[9]

- La norme JPEG doit être très proche des techniques nouvelles de compression, en terme de taux de compression (un excellent taux de compression sur des images ayant les points codés de 6 à 24 bits.), de qualité de restitution et temps de calculs.
- JPEG doit pouvoir compresser n'importe quel type d'images réelles. (peut gérer différentes formes d'expression des couleurs comme RGB, CYM ou les niveaux de gris).
- L'algorithme doit être implémenté sans trop de problèmes sur une grande gamme de CPUs, et sur des cartes spécialisées.
- Le codage doit pouvoir être séquentiel, progressif et sans pertes.
- JPEG est destiné à des images de type photographique. Les images composées de vecteurs, de droite ou d'objet géométrique, ne seront pas compressées avec une excellente efficacité.

JPEG est prévu pour supprimer certaines informations, invisible à l'œil nu, dans une image.

Exemple

La figure suivante montre la compression JPEG:

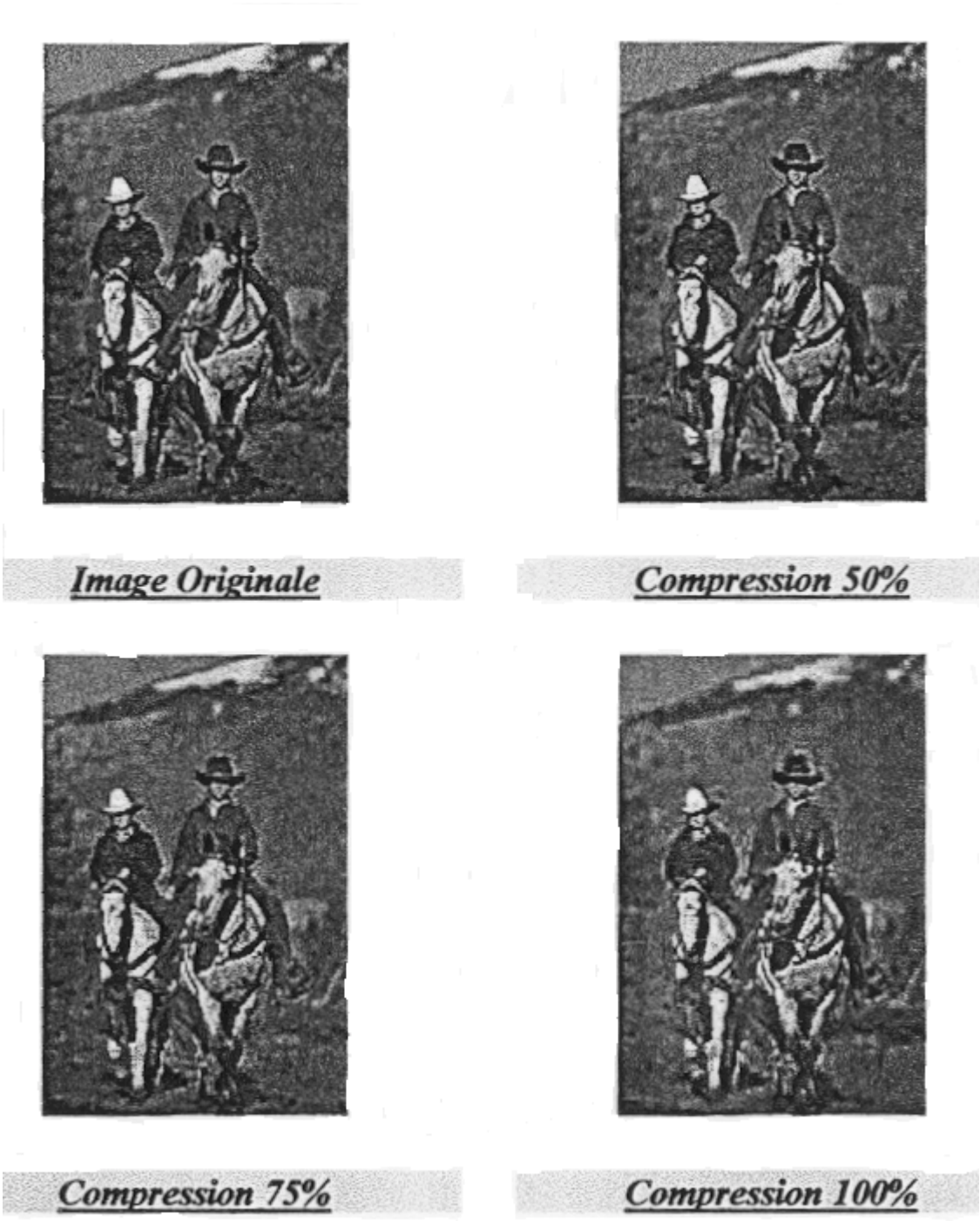

Figure 2.5 Exemple de Compression JPEG

2.3.2.5 La Norme JPEG 2000

Il faut d'abord rappeler que le format JPEG actuel a été développé, il y a plus de 10 ans.

Les technologies ne sont plus les mêmes aujourd'hui car la recherche a évolué et d'autre part, les besoins ne sont plus les mêmes non plus. Il faut répondre

au développement de la photo numérique, du Web et du commerce électronique par exemple.

On a aujourd'hui besoin

- d'un format ouvert permettant l'échange d'images entre des logiciels et des équipements différents.
- d'un format utilisant des techniques de compression modernes dans le but de transmettre des contenus plus riches en haute résolution.
- d'un format qui permette d'obtenir, à partir du même fichier, la même image à des résolutions différentes pour tenir compte de la bande passante disponible.

Un tel programme est difficile à réaliser. Le groupe de travail ISO JPEG 2000 y travaille depuis 1998.

On peut tout de même donner un premier aperçu de ce que devrait être le format JPEG 2000 en présentant 3 de ses caractéristiques principales:

- Il sera basé sur la technologie des Ondelettes. Le résultat pratique de ce choix devrait être une amélioration modeste mais réelle du taux de compression (de l'ordre de 20 %) et une qualité supérieure de restitution par rapport à ce que permet le JPEG actuel. Les coefficients d'Ondelettes sont plus faciles à stocker que les blocs résultants de l'application de la DCT.
- Il permettra à l'utilisateur de déterminer jusqu`à quel point les détails de l'image l'intéressent en lui donnant le choix de la résolution de restitution. La récupération sans pertes de l'image devrait être possible.
- Il permettra une meilleure restitution des couleurs grâce à l'utilisation des profils ICC. On pourra aussi bien enregistrer des images RVB que des images CMJN [10].

2.3.2.6 La compression par Ondelette

Les ondelettes c'est d'abord une théorie mathématique récente d'analyse du signal développée dans les années 80. On peut considérer qu'il s'agit d'une extension de l'analyse de Fourier.

Elles sont des fonctions générées à partir d'une fonction mère Ψ, par dilatations et translations. Dans le cas mono-dimensionnel, la fonction s'écrit : [14], [22]...

$$\Psi_{a,b}(x)=\frac{1}{\sqrt{|a|}}\Psi\left(\frac{x-b}{a}\right)$$

Où l'indice a représente un facteur d'échelle et l'indice b est un facteur de translation.

On a un signal continu et on le décompose en une série de nombres qui décrivent des courbes qui s'additionnent pour reconstruire le signal.

L'intérêt de cette théorie est au départ l'analyse des signaux et elle a déjà de nombreuses applications [22].

Nous allons maintenant donner une idée de la méthode de compression qui utilise les ondelettes. Elle est résumée par le schéma (figure 2.6) [17].

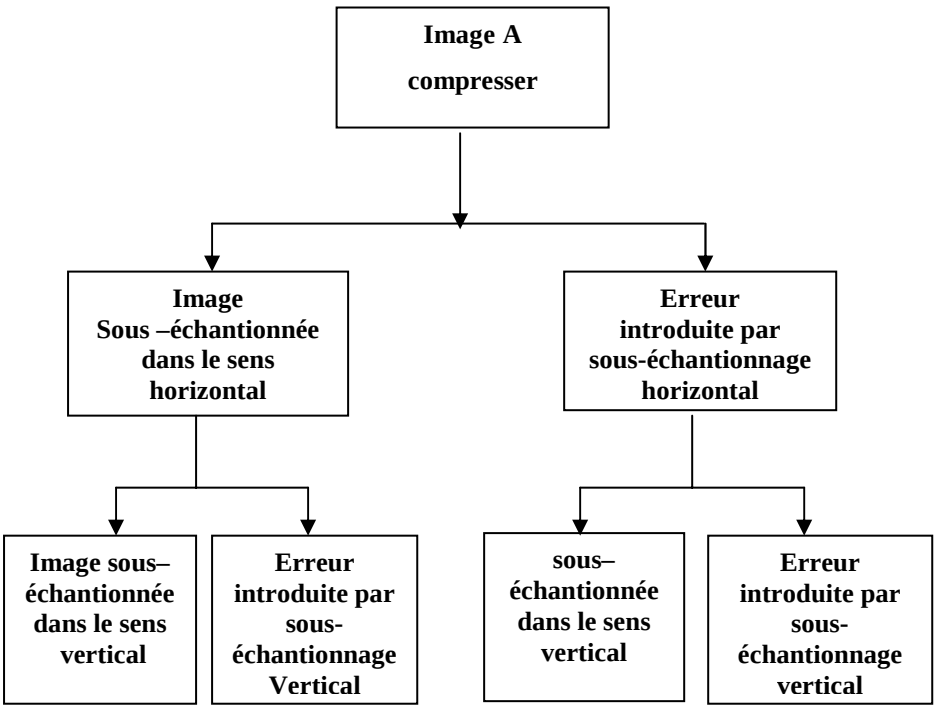

Figure 2.6 Codage par Ondelette

Algorithmes Pyramidal de Burt & Adelson

- On fait un sous-échantillonnage de l'image dans le sens horizontal.
- On calcule l'erreur entre l'image originale et l'image sous-échantillonnée dans le sens horizontal.
- Pour chacune des 2 images obtenues, on fait un sous-échantillonnage dans le sens vertical.
- Pour chacune des 2 images obtenues, on calcule l'erreur dans le sens vertical.
- On obtient une image dont la résolution est divisée par 2 et 3 images qui codent les erreurs entre l'image originale et l'image sous-échantillonnée
- On répète cette transformation un certain nombre de fois puis on effectue une quantification.
- On abandonne les détails inférieurs à un certain niveau et on code les valeurs restantes.

En conclusion on retiendra les points suivants :

La compression par Ondelette est une technique récente qui donne de très bons résultats, même avec des taux de compression élevés (plus de 90%). Cette méthode reste encore marginale par rapport à l'utilisation de JPEG, malgré ses avantages.

- ❖ Cette méthode permet de prévoir le-taux de compression contrairement au JPEG.
- ❖ Elle n'entraîne pas d'effet de mosaïque.
- ❖ L'algorithme est plus simple et plus souple que JPEG et donc plus rapide.
- ❖ Il est possible d'avoir des images très compactes (de l'ordre du ko !).
- ❖ Une image compressée par les Ondelettes peut être décompressée de deux manières différentes :
  - ✓ sa résolution est fixe mais sa taille augmente progressivement.
  - ✓ sa taille est fixe mais sa résolution augmente progressivement

2.4 Critères du choix de la méthode de compression

Les points importants dans le choix d'une méthode de compression d'images sont : [18]

- ❖ Le type d'image à compresser (photographies, dessins en couleur, plans ou pages de texte numérisés) ?
- ❖ L'objectif de rapport qualité / ratio de compression (niveau des pertes dues à la compression versus le poids des images compressées) ?
- ❖ La vitesse de compression et de décompression ?
- ❖ Les modes de fonctionnement optionnels (mode progressif zoom...)

2.5 Conclusion

Après avoir présenter les techniques et les algorithmes de compression d'image, nous pouvons tenir les points suivants :

Nous distinguons deux types de compression suivant leur usage :

- Les données informatiques qui doivent rester identiques à leur original (textes, programmes informatiques, ...) utilisent la compression sans perte (RLE, LZW, Huffman, ...).
- Les données dont la qualité se limite aux perceptions humaines (images, vidéos, sons, ...) utilisent la compression avec pertes (JPEG, ondelettes, MPEG, ...).

Les différents algorithmes de compression sont choisis en fonction de :

- Leur taux de compression (rapport de la taille du fichier compressé sur la taille du fichier initial)
- La qualité de compression (sans/avec perte et alors pourcentage de pertes)
- La vitesse de compression et de décompression.

Le chapitre suivant est consacré aux éléments mathématiques de la méthode de compression par fractales.

Chapitre 3

La théorie Fractales : outils et fondements mathématiques

Nous allons étudier dans ce chapitre quelques notions et concepts concernant la théorie des Fractales, et nous présentons la théorie IFS.

3.1 La Géométrie fractale

La géométrie fractale comme développée initialement par Mandelbort a eu l'impact majeur en modélisation des formes et de processus naturels [23].

Par exemple un arbre est représenté d'une manière approchée dans la géométrie euclidienne grâce à des primitives géométriques simple (comme des cylindres ou des sphères), alors qu'en géométrie fractale, la forme des feuilles et des branches est plus proche de la réalité [16].

- **Fractale**

Est un adjectif mathématique se dit d'objets mathématiques dont la création ou la forme ne trouve ses règles que dans l'irrégularité ou la fragmentation, et des branches des maths qui étudient de tels objets [24], [25].

Malgré le peu de résultats théoriques disponibles (théorèmes...), les applications ne se font plus attendre, dans la simulation ou la modélisation, car beaucoup d'objets naturels ont une structure fractale (Flocons de neige, choux-fleurs, arbres, nuages).

3.1.1 L'image Fractale

Une image fractale est une figure géométrique induite par un processus répétitif dans la limite de résolution graphique de l'écran d'un ordinateur, sous l'action d'opération de translation, de rotation et de changement d'échelle [26].

3.1.2 Objet fractal

C'est un objet naturel dont la forme est extrêmement irrégulière et de plus (éventuellement) interrompue et fragmentée [16], [27].

3.1.2.1 Auto-similarité

Un objet est dit auto-similaire si sa forme est la même à n'importe quel grossissement. Un détail de l'objet une fois agrandie, reproduit exactement une partie plus grande de l'objet. Cependant il existe des objets fractals qui ne sont pas exactement auto-similaire, l'objet ne contient pas une transformation de lui-même c'est ce qu'on appel l'auto similarité locale [16].

La notion d'auto similarité revient dans plusieurs contextes car les objets naturels sont auto-similaire (ou presque).

La figure 3.1 illustre cette notion d'auto similarité par un exemple de triangle de Sierpinsky : ce triangle est constitué de trois triangles est lui-même est constitué de trois autres triangles qui sont une réduction d'un facteur [21].

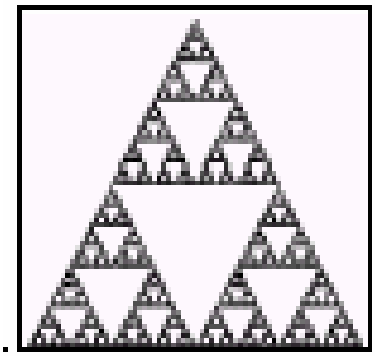

Figure 3.1 Triangle de Sierpinski

- **Flocon de Von Koch**

En 1904, Helge Von Koch (1870-1924 - Suède) publie l'article : « Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire » qui décrit la ligne actuellement connue sous le nom de 'flocon de Von Koch'.

La méthode :

Pour tracer cette courbe, il faut :

- Tracez un triangle équilatéral
- Remplacer le tiers central de chaque côté par une pointe dont la longueur de chaque côté égale aussi au tiers du côté.
- Recommencer cette construction sur chaque côté des triangles ainsi formés.

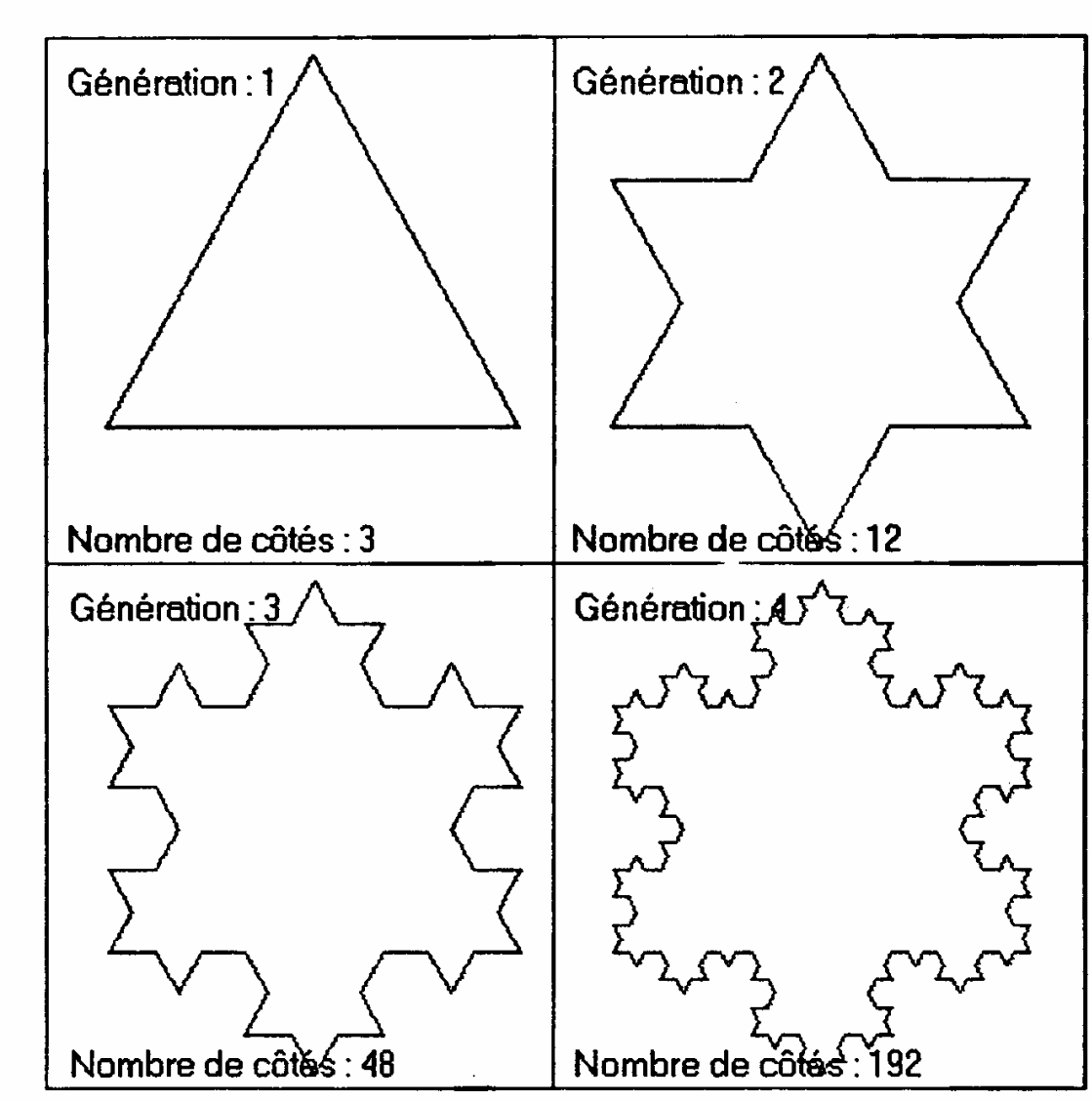

Figure 3.2. Flocon de Von Koch

Les objets géométriques engendrés par une application itérée, telle que la courbe de Von koch, sa construction nécessite un initiateur, un générateur.

Elle est soumise à une application itérée qui est définie par le générateur [28].

3.1.2.2 Dimension fractale

La définition de la dimension fractale découle de la définition de l'auto-similarité.

Un ensemble A est dit auto-similaire s'il est l'union de *N* copies distinctes de lui-même, chacune étant mise à l'échelle avec un rapport d'homothétie *r* pour toutes les coordonnées. La dimension fractale ou d'homothétie de A est alors donnée par la relation :

$$1=Nr^{D} \quad \text{ou} \quad D=\frac{\log N}{\log(1/r)}$$

Les surfaces fractales naturelles, en général, ne possèdent pas cette auto similarité déterministique; cependant, elles montrent une auto-similarité statistique. Ceci étant, elles sont composées de *N* sous-ensembles distincts mis à l'échelle avec un rapport r à partir de l'ensemble original et identiques statistiquement par rapport à l'original. La dimension fractale pour ces surfaces est aussi donnée par l'équation précédente [23], [29], [30].

3.1.2.2.1 Dimension Topologique

La dimension topologique d'un ensemble totalement discontinu est toujours nulle.

La dimension topologique d'un ensemble F est n si n'importe quel voisinage arbitrairement petit de n'importe quel point de F un bord de dimension topologique n-1.

Typiquement, un intervalle de R est de dimension 1, une zone compacte du plan est de dimension 2,... [21], [29].

3.1.2.2.2 Dimension de Hausdorff

Considérons l'espace métrique $\left(R^2,d\right)$. $H(R^2)$ désigne l'espace dont les éléments sont les sous-ensembles compacts non vides de R^2 différents de l'ensemble vide.

La distance du point x élément de R2 à l'ensemble B élément de $H(R^2)$, notée $d(x,B)$, est définie par : [14]

$$d(x,B)=\min\{d(x,y):y\in B\}.$$

La distance de l'ensemble A élément de $H(R^2)$ à l'ensemble B élément de $H(R^2)$, notée $d(A,B)$, est définie par :

$$d(A,B)=\max\{d(x,B):x\in A\}.$$

La distance de Hausdorff entre deux ensembles A et B éléments de $H(R^2)$, notée $h_d(A,B)$, est définie par :

$$h_d(A,B)=\max\{d(A,B),d(B,A)\}.$$

3.2 Systèmes des fonctions itérées IFS

Les IFS (iterated fonction system) ont été introduits en 1985 par M.F Barnsley et S.Denko.

3.2.1 Principe de la machine à copier

Un exemple simple : La Multiple Réduction Copy Machine (MRCM).

C'est une métaphore: imaginez une photocopieuse qui réduit la taille d'une image de 50% puis la recopie en 3 exemplaires aux sommets d'un triangle équilatéral (la taille de l'image originale est donc la même que celle du "triangle" des copies). La figure 3.3 ci-dessous illustre l'idée.

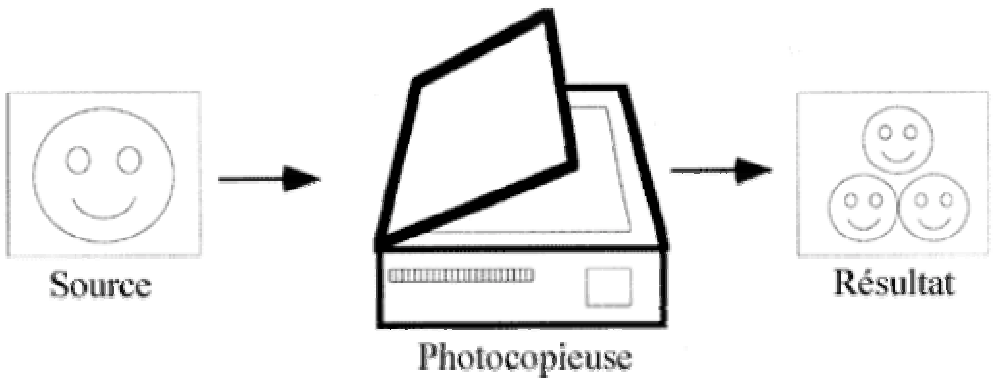

Figure 3.3 La Multiple Réduction Copy Machine

Imaginez ensuite qu'à chaque cycle de la copie, la photocopieuse fonctionne en feedback, c'est à dire que le document qui vient de sortir est remis en entrée de la photocopieuse.

Alors, à force de recopier avec les mêmes transformations (ici division par 2 de l'échelle puis recopie de la source 3 fois), la sortie convergera vers ce que l'on appelle un attracteur, c'est à dire une image qui quand on la remet en entrée de cette photocopieuse donne en sortie la même image.

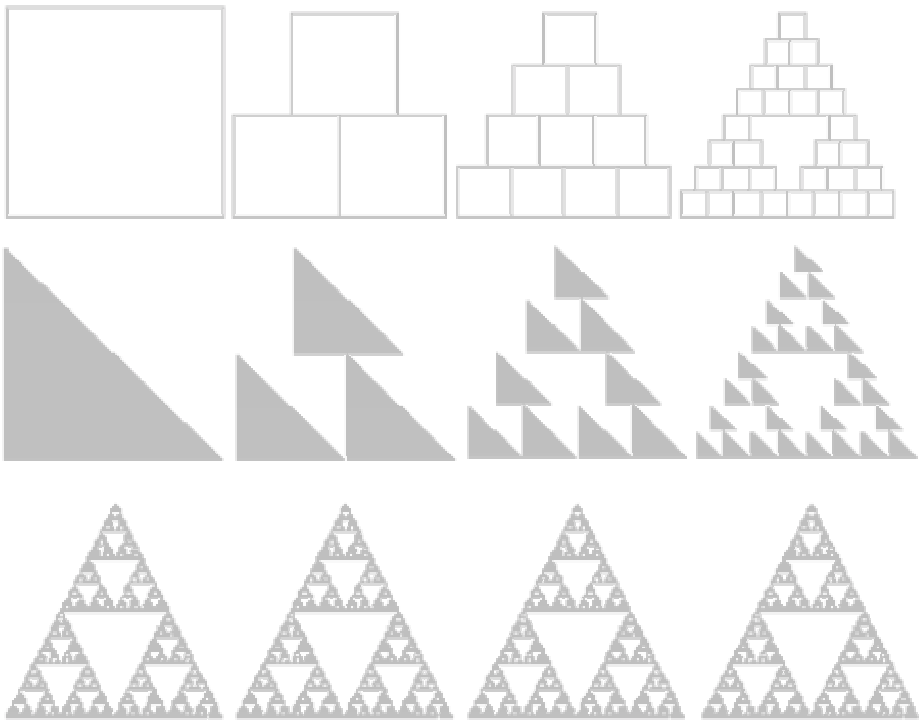

Figure 3.4

- **Indépendance de l'attracteur vis-à-vis de l'image d'origine**

Nous considérons deux images totalement différentes, et appliquons à chacune d'elles simultanément les mêmes transformations. Comme le montre (la figure 3.5) ci-dessous, les deux images (a et b) qui au départ étaient dissemblables donnent lieu au même attracteur (c).

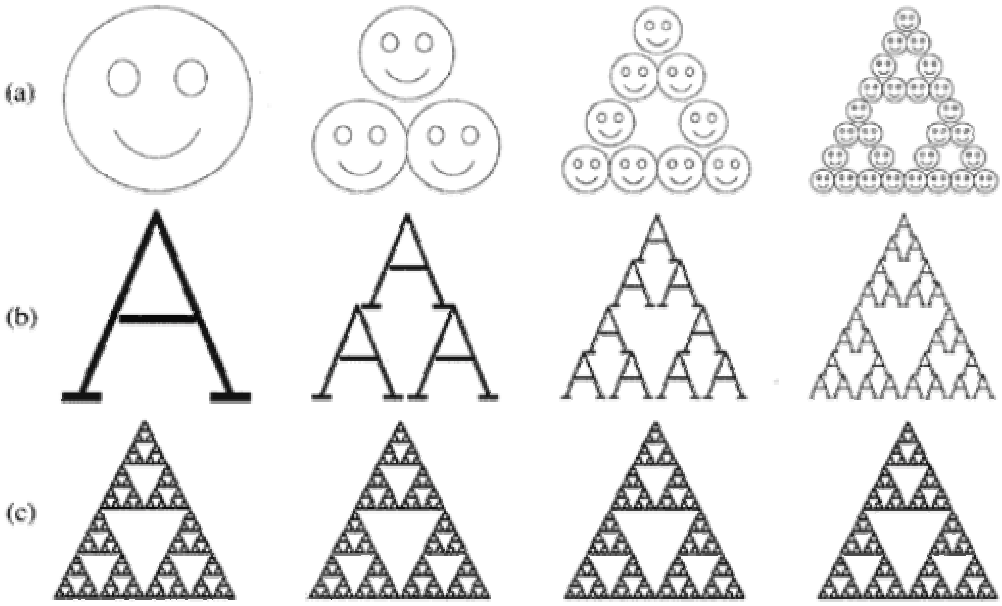

Figure 3.5

En fait, plus on réalise de copies, plus l'image d'origine disparaît pour ne devenir qu'un point. Mais ce point existe en plusieurs copies agencées d'une certaine manière que définit la transformation utilisée.
Ainsi, un attracteur ne dépend que de la transformation utilisée.
En pratique, ces transformations sont des applications affines.

3.2.2 Transformation affine

C'est un premier outil pour la génération de scènes fractales en utilisant les IFS. C'est une combinaison d'un ensemble de transformations linéaires, constituées de rotation, changement d'échelle et de translation l'effet d'une telle fonction sur un point P(x , y), est de transformer ses coordonnées en une autre position représentée par le point P'(x' , y') [16].
Cette transformation est définie par :

$$W:\Re\rightarrow\Re \quad p(x,y)\mapsto p'(x',y')$$

Avec :

$\exists(a,b,c,d,e,f)\in\Re^6$:

$$W\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix}$$

Où :

- Les paramètres a, b, c, d produisent une rotation et un changement d'échelle
- Les paramètres e et f définissent les facteurs de translation du point sur lequel on opère.

En appliquant un ensemble de transformations à une figure géométrique, celle-ci va subir une translation vers une nouvelle position, puis va être soumise à une rotation et enfin réduite à une taille inférieure.

En prenant compte du niveau de gris dans l'image, on ajoutera une autre dimension, et la transformation devient ainsi :

$$W\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & g \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e \\ f \\ h \end{pmatrix}$$

où

- g contrôle le contraste.
- h contrôle la luminosité de la transformation.
- z représente le niveau de gris.

3.2.3 Transformation Lipchitzienne

Soit $W:\Re^2 \to \Re^2$ une transformation définie sur l'espace métrique $(\Re^2, d)$.

d désigne la distance entre deux points de $\Re^2$. La transformation ω est dite lipchitzienne si : [14]

$$\exists s \in \Re^{+*} : d(\omega(x), \omega(y)) \leq s.d(x,y) \qquad \forall x, y \in \Re^2$$

s est appelée coefficient de lipchitz de W.

3.2.4 Transformation contractive

Une transformation lipchitzienne est dite contractive si, le coefficient s est strictement inférieur à 1, ($0 < s < 1$) [16].

Cette transformation contractive appliquée à deux points va les rapprocher. Cette définition est absolument générale, elle s'applique à tout espace métrique (espace sur lequel on peut définir une distance d) [20].

Par exemple pour tous deux points $P1(x_1,y_1)$ et $P2(x_2,y_2)$, la distance est calculée comme suite :

$$d(P1,P2)=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

3.2.5 Point fixe

Soit $\Re^2$ un espace métrique, et $W:\Re^2\to\Re^2$.

La transformation contractive W possède un point fixe unique $x_f\in\Re^2$, tel que $W(x_f)=x_f$.

Pour tout point x élément de $\Re^2$, la séquence $\{w^{on}(x):n=0,1,2,...\}$ converge vers x_f :

$$\lim_{n\to\infty} w^{on}(x)=x_f \quad \forall x\in\Re^2.$$

3.2.6 La définition d'un IFS

Un IFS consiste en collection de transformations contractantes :

$\{W_i:\Re^2\to\Re^2/i=1...n\}$

Cette collection de transformations définit une fonction :

$$W=\bigcup_{i=1}^{n} w_i$$

La fonction W n'est pas appliquée au plan, mais à des ensembles de points du plan.

Si Les w_i sont contractantes dans le plan, alors W est contractante dans l'espace de l'ensemble de points du plan, (démonstration de Hutchinson) [16], [20].

3.2.7 Attracteur d'un IFS

Soit un IFS $\{\Re^2; w_i; i=1,...N\}$.il a été démontré que l'opérateur W: $H(\Re^2)\rightarrow H(\Re^2)$ défini par: :

$$W(B)=\bigcup_{i=1}^{N}\omega_i(B),\ \ \forall B\in H(\Re^2)$$

Est contractant et a pour facteur de contraction celui de l'IFS. L'opérateur W possède un unique point fixe x_f donnée par :

$$x_f=W(x_f)=\lim_{n\rightarrow\infty} w^{on}(X),\ \ \forall X\in H(\Re^2)$$

L'objet x_f est aussi appelé attracteur de l'IFS. Il est invariant sous la transformation W et est égal à l'union de N copies de lui-même transformées par $\omega_i\cdots\omega_N$ (Figure 3.6). L'objet invariant est dit auto-similaire, ou auto-affine lorsque les transformations élémentaires ω_i sont affines [14].

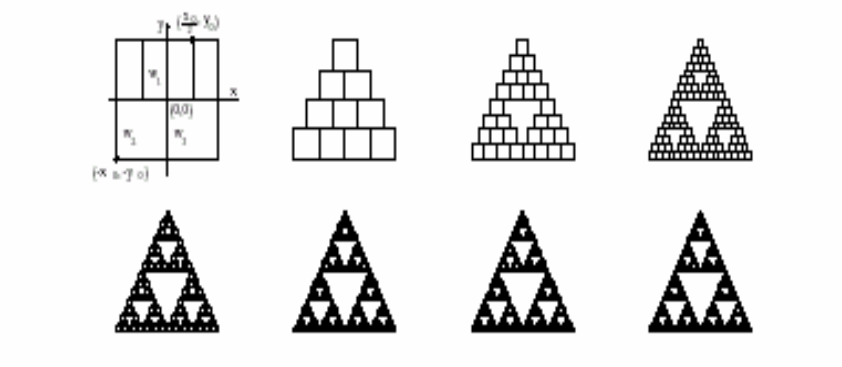

Figure 3.6 illustre le point fixe d'une transformation affine contractive par la Répétition de l'application W

3.2.8 Les IFS comme outils de compression

Comme l'a suggéré M. Barnsley, la compression fractale consiste non pas à enregistrer les informations propres de l'image, mais plutôt les transformations

(au sens vu précédemment) nécessaires à la constitution de l'image en tant qu'attracteur. Connaissant cette suite de transformations, l'image pourra être reconstituée en partant de n'importe quelle image source (par exemple une image entièrement noire) car celle-ci devenant un point parmi d'autres.

Voyons à présent la taille nécessaire au codage d'une transformation (on verra plus tard que cela n'est pas suffisant pour coder une image quelconque complètement).

Comme il a été vu précédemment, chaque transformation est définie à partir de 6 réels qui sont codables sur 32 bits. II suffit donc de 192 bits pour stocker une transformation.

Pour compresser l'image de la fougère ci-dessous (figure 3.7), comme il ne faut que 4 transformations (on ne prend pas encore en compte les transformations liées aux niveaux de gris), il suffira de 4 x 192 bits = 768 bits (96 octets) pour stocker l'image alors qu'il faut environ l2kbits pour stocker l'image avec la résolution actuelle.

Alors que l'image (de fougère de taille 256*256) sous forme de pixels requiert beaucoup plus de mémoire au moins 65536 bits.

On voit que le stockage des images sous forme de collection de transformations contractantes amène directement la compression puisque le stockage d'une image sous forme de pixels nécessite beaucoup de place mémoire que de stocker l'image sous forme de collection de transformations contractantes.

3.2.9 Théorème du collage

Ce théorème fournit une borne supérieure à la distance de Hausdorff h_d entre un point x inclus dans $H(\Re^2)$ et l'attracteur x_f d'un IFS [14], [16].

Soit l'espace métrique complet $(\Re^2,d)$ Soit un point $x\in H(\Re^2)$.

et un IFS $\{\Re^2,\omega_1,\omega_2,...,\omega_n\}$ avec le réel $0\leq s<1$ pour facteur de contraction. Alors nous avons la relation :

$$h_d(x,x_f)\leq\frac{1}{1-s}h_d\left(x,\bigcup_{i=1}^{n}\omega_i(x)\right)$$

Le théorème indique que s'il est possible de transformer un objet x de manière à vérifier $x \approx W(x)$ tout en s'assurant que W est contractant, alors le point fixe x_f de l'opérateur W est proche de x [20].

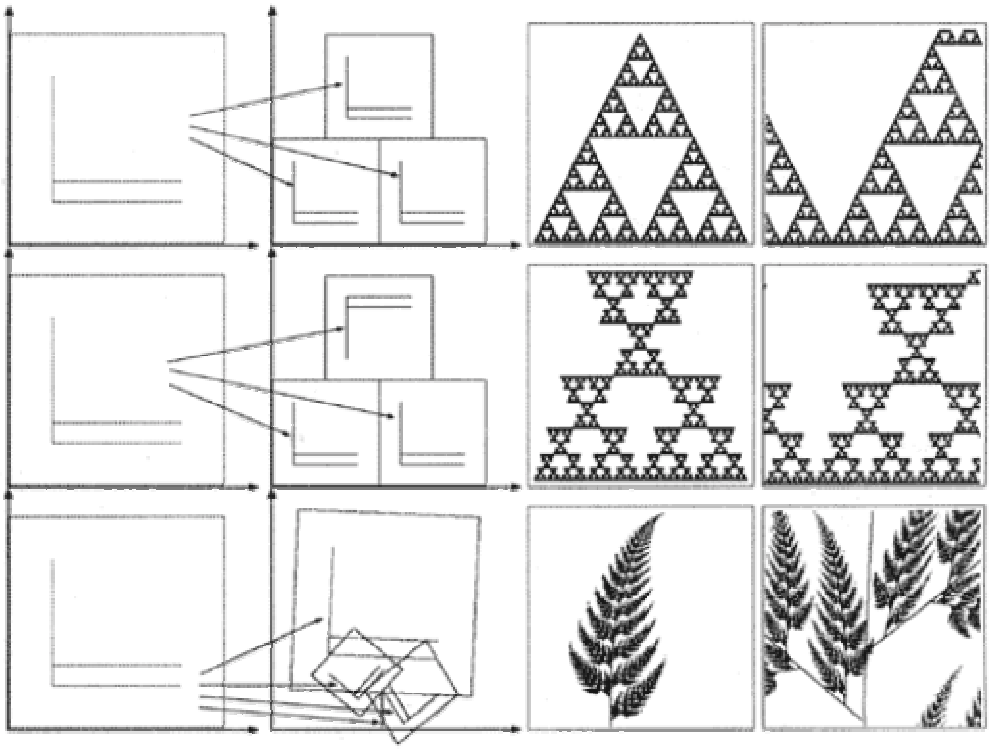

***Figure** 3.7*

3.2.10 Transformation finalement contractante

Soit une transformation Lipchitzienne ω. Cette transformation est dite finalement contractante, s'il existe un entier n (nombre d'itérations) tel que la transformation ω^{on} est contractante [14].

n est appelé exposant de contraction.

3.2.11 Théorème du collage généralisé

Considérons l'opérateur W finalement contractant, avec pour exposant de contraction l'entier n ; il existe alors un unique point fixe [14], [21] $x_f \in \Re^2$ tel que :

$$x_f = W(x_f) = \lim_{k \to \infty} W^{ok}(x) \quad \forall x \in \Re^2$$

Dans ce cas :

$$h_d(x, x_f) \leq \frac{1}{1-s} \frac{1-\sigma^n}{1-\sigma} h_d(x, W(x))$$

Où :

σ est le facteur de Lipschitz de W.

s est le facteur de contraction de W^{on}.

Ce théorème réellement la clé ou le chemin vers le processus de compression fractale.

3.3 Les PIFS

Informellement, il s´agit d´une collection de fonctions qui ont comme domaine non plus l´espace entier mais seulement une partie. En utilisant l´image de la photocopieuse, la photocopieuse PIFS ne recopierait pas l´image toute entière mais seulement des morceaux de celle-ci à différents endroits et avec différentes orientations. Les PIFS ont été définis par Jacquin.

- **Défintion d´un PIFS**

Un PIFS est un ensemble de transformations contractantes, définies sur l´espace (R^2,d) : $\{ \omega_i : D_i \rightarrow R^2, i = 1, \ldots, n\}$

La définition d´un PIFS est indépendante du type de transformation utilisé, mais on se limitera aux transformations affines. Le niveau de gris ajoute une troisième dimension z, donc les transformations w_i sont de la forme [31] :

$$\omega_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} \quad avec\ i = 1,2,\ldots,n$$

Où s_i contrôle le contraste et o_i la luminosité de la transformation.

Il est pratique d´écrire :

$$v_i(x, y) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

comme une image est modélisée par une fonction $f(x,y)$, on peut appliquer w_i à une image f par $w_i(f) = w_i(x, y, f(x, y))$. Alors v_i détermine comment les domaines, partitions de l´original, sont placés sur la copie, tandis que s_i et o_i déterminent le contraste et la luminosité de la transformation.

3.4 Conclusion

Après avoir analyser la théorie Fractale, et ses éléments mathématiques, nous abordons dans le chapitre suivant cette théorie qui fait l'objectif d'une partie de notre travail en tenant compte l'application de cette dernière dans la compression d'image.

Chapitre 4

Fractale Comme Méthode de Compression (Encodage et Décodage)

La majorité des algorithmes de compression autres que la compression fractale ne considère l'image que dans sa globalité, en ne l'utilisant que comme une information binaire pure.

La compression fractale, quant à elle, considère toute image comme un assemblage d'attracteurs pour ensuite la compresser en ne codant que chaque transformation associée à chaque attracteur. Donc, si l'image a été partitionnée en N partitions (ou attracteurs), il y aura à coder exactement N transformations dans le fichier final. Et donc, si une transformation fractale occupe b bits, la taille du fichier final sera donc égale à Nxb bits. On peut dès lors constater que plus il y a de partitions, plus la taille du fichier final sera grande.

Pour obtenir une compression optimale, l'algorithme devra donc minimiser le plus possible le nombre de partitions nécessaires au codage de l'image.

4.1 La Méthode Fractale

La problématique de la compression fractale : pour une image donnée A, est de trouver un IFS qui a comme attracteur, l'image originale. C'est-à-dire, trouver une collection de transformations affines W qui après une application récurrente sur une image quelconque, génère l'image originale comme attracteur qu'on veut compresser [16].

Les redondances dans l'image sont des parties de l'image qui se ressemblent entre elles.

Les images naturelles ne sont pas exactement identiques entre elles (redondances).

La même figure (voir figure 4.1 Lena) ne contient pas le type de similarité que l'on peut trouver dans les images générées par des IFS où un motif est reproduit par une transformation [15].

4.1.1 L'auto- similarité dans l'image

L'image n'apparaît pas comme contenant des transformations d'elle-même, mais en fait cette image contient différentes sortes de similarités [15].
(La figure 4.2) montre des exemples de régions de l'image de Lena qui sont similaires à des échelles différentes : une portion de son épaule recouvre une région qui est presque identique, et une portion du reflet du chapeau dans le miroir est similaire (après transformation) à une autre partie de son chapeau.

Figure 4.1 Lena

Figure 4.2 les auto-similarités dans Lena

La différence avec le type d'auto similarités rencontrées dans le triangle de Sierpinsky (figure 3.1) est que plutôt que d'avoir de copies entières d'elle-même (après certaines transformations affines appropriées), nous avons de copies de parties (transformés de parties) d'elle-même [32].

Ces parties n'étant pas, sous certaines transformations affines, exactement des copies identiques d'elle-même, il faut donc permettre une certaine erreur dans notre

représentation d'une image comme un ensemble de transformations ce qui implique que l'image que nous encodons comme ensemble de transformations ne sera pas une copie identique à l'originale, mais plutôt une bonne approximation de celle-ci.

Donc, l'établissement d'un bon code fractal revient à la bonne exploitation des redondances au sein de l'image.

4.1.2 Pourquoi cette compression est-elle fractale ? [14]

Le principe de compression d'image sur lequel s'appuie cette technique peut être dit fractal car l'ensemble des transformations sous la forme duquel l'image est codée est très proche de la MRCM (voir chapitre 03). Ceci a plusieurs implications :

Comme les IFS (la célèbre fougère de Barnsley par exemple), l'image décodée présente des détails à toutes les échelles : si on agrandit l'image d'un facteur k, l'attracteur de l'image sera lui aussi agrandi du même facteur k.

L'image décodée n'a pas de taille prédéfinie : sa taille est indépendante de celle de l'image initiale et tous les détails supplémentaires sont générés automatiquement par le procédé de restitution. Ces nouveaux détails (si l'image décodée est plus grande que l'image initiale) ne sont évidemment pas réels, dans le sens qu'ils ne proviennent pas de l'image initiale, mais dans certains cas, ces détails sont réalistes pour de faibles agrandissements.

4.2 Procédure et schéma général d'un codeur - décodeur

La procédure générale concernant la compression/décompression fonctionne de la manière suivante : (figure 4.3) [21]

4.2.1 La compression

D'après le schéma, on peut résumer les différentes étapes qui interviennent dans la création de l'image compressée comme suit : [15]

Début :

1. Ouvrir le fichier image et donner la distorsion RMS
2. Donner : $T_{\min}$, $T_{\max}$;
3. Classification des blocs domaines et construction de la librairie domaine ;
4. Partitionnement QuadTree et récursif de l'image jusqu'à la taille $T_{\max}$ (donné) ;

5. Chaque bloc range de l'image passe par les traitements suivants :
   - Classifier le bloc range ;
   - Chercher dans la classe du bloc range le bloc domaine telle que la distorsion calculée sera inférieure à la distorsion donnée ;
   - S'il y a un succès alors le code (la transformation) du bloc range est stocké
   - S'il y a un échec :
     a. Si la taille du bloc range est supérieure à la taille $T_{\min}$ (donné) alors il est décomposé a nouveau ;
     b. Si la taille du bloc range est inférieure à la taille $T_{\min}$ ce bloc composé avec le meilleur bloc domaine et son code sera stocké ;
6. Libérer la librairie domaine ;
7. Fermer le fichier ;

Fin. (Les détaille des étapes en l'annexe B).

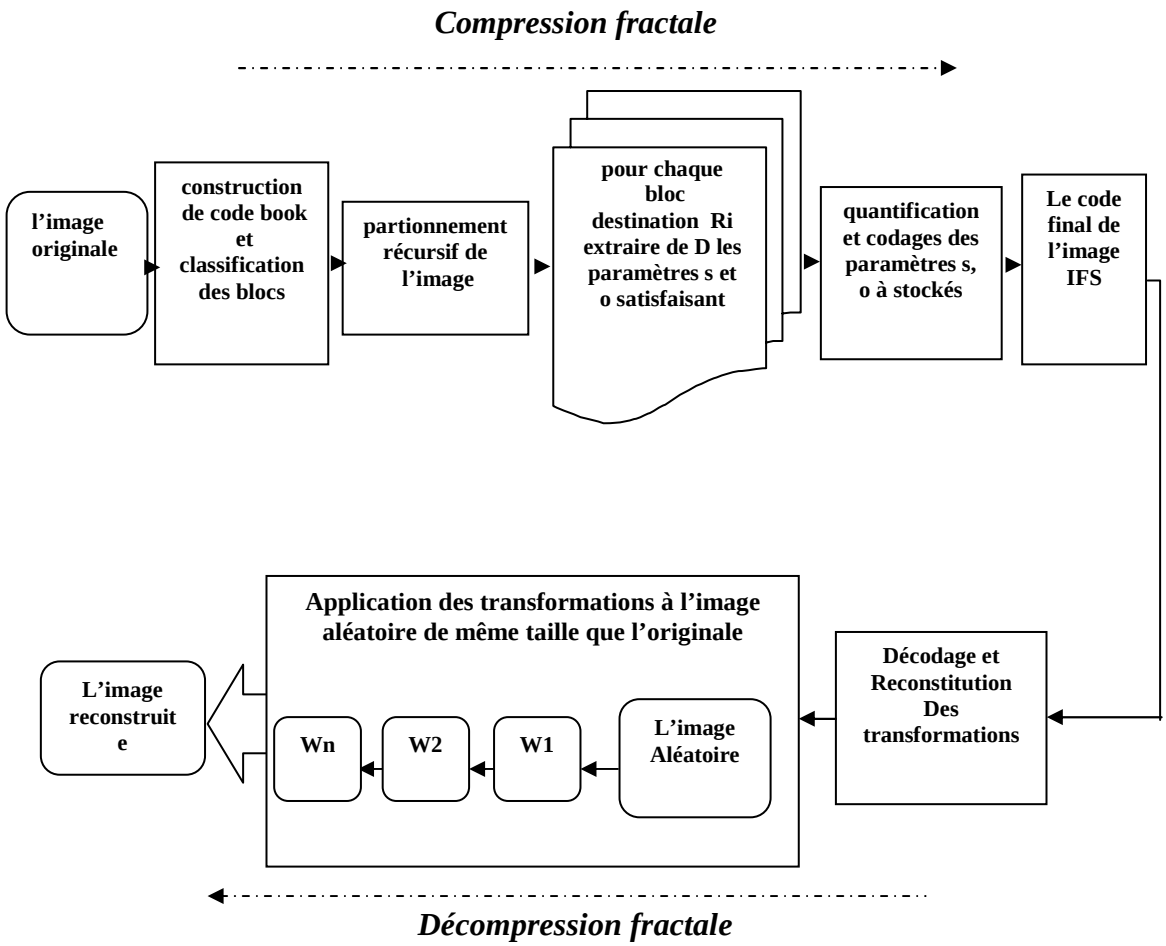

Figure 4.3 Le schéma général d'un codeur – décodeur

4.2.2 La décompression

On peut résumer aussi les différentes étapes de la décompressée comme suit : [15]

Début :

1. Ouvrir le fichier de code obtenu par l'algorithme de compression ;
2. Charger une image initiale de même taille que l'image originale ;
3. Lire et appliquer le code fractal ;
4. Reconstruire l'image ;
5. L'image construite devient l'image initiale ;
6. Tant que le nombre d'itérations n'est pas terminé, refaire (3), (4) et (5).
7. Afficher l'image reconstruite ;

Fin

4.2.2.1 Le processus de décompression

Lors de l'étape de compression, nous avons vu quel est était le schéma de la décompression (voir Figure 4.7). Rappelons cependant son principe.

Pour décompresser un fichier contenant une image compressée par la méthode fractale, il suffit de construire tous les attracteurs constructibles à partir des informations contenues dans l'en-tête du fichier contenant l'image compressée (à savoir la paire de fonctions d'intensité). En suite, il faut lire les informations concernant chaque bloc-fils, déterminer après un nombre d'itérations suffisantes (la convergence est rapide, ce qui permet d'obtenir des vitesses de décompression importantes par rapport à la phase de compression) le nouveau bloc qui lui correspond. Selon l'indice de sa rotation, on opère la rotation opposée et l'on obtient alors un bloc d'autant plus similaire à l'original que la correspondance énoncée en phase de compression est importante.

C'est en rassemblant les blocs-enfants obtenus que l'on reforme les plans codés sur 8 octets, puis, si nécessaire, l'image sur 24 plans si c'est une image en millions de couleurs. Les résultats successifs des phases de décompression d'une image compressée par ce procédé sont proposés sur cette image.

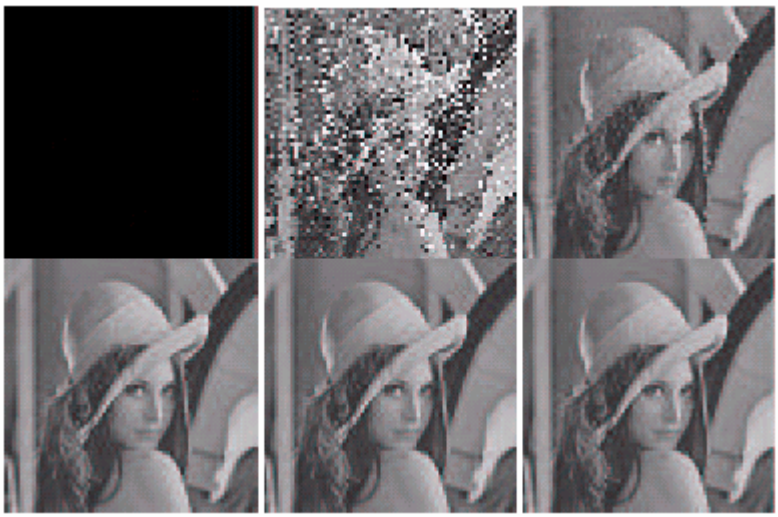

Figure 4.7 Différentes phases de décompression d'une image compressée par Le procédé Fractal. on voit que la convergence est rapide.

4.3 Quelques méthodes de partitionnement

Différents types de partitionnement de l'image pour coder le mieux les similarités inter-blocs dans la compression à base des fractales, de façon à minimiser le nombre de transformations locales.

Nous distinguerons trois classes de partitionnement [14], [1]

1. les partitionnements rigides.
2. les partitionnements semi-rigides.
3. les partitionnements souples.

- **Utilité et rôle du partitionnements en codage**

Les modèles de partitionnement sont des outils qui peuvent être utiles en analyse et en traitement d'images notamment en codage. Le codage d'une image à l'aide de partitionnement consiste à approcher la surface des niveaux de gris de l'image dans chacun des blocs de la partition au moyen de fonctions bidimensionnelles [21], [14].

L'objectif est dans ce cas de construire une partition dont chacun des blocs vérifie un prédicat donné. Lorsque le but est de compresser l'image, un

compromis doit être trouvé entre, d'une part la complexité du modèle de partitionnement et d'autre part celle du codage.

4.3.1 Partitionnement rigide

Un partitionnement est qualifié de rigide s'il ne s'adapte pas moindre coût aux formes des objets présents dans l'image.

4.3.1.1 Partition QuadTree

Le but de cette technique [14] est de subdiviser l'image en n parties disjointes ayant pour réunion l'image en entier de façon à obtenir un RMS (Root Mean Square error) correct. On suppose, si N est le pas de la subdivision que l'image a des dimensions multiples de N.

Méthode :

On choisit N, en général 8, et on subdivise l'image en carrés adjacents de taille Nx N. Pour chaque carré de taille N' x N', que l'on appelle R_i (au début N = N') on cherche un W_i c'est à dire une application contractive envoyant en autre carré D_i sur R_i.

Si aucun W_i n'est pas satisfaisant, on subdivise R_i en 4 carrés adjacents de taille (N'/2)x(N'/2) et on recommence l'étape 2, car les chances de trouver un W_i correct augmentent quand la taille du bloc image diminue.

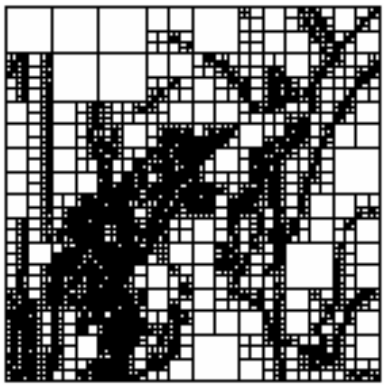

Figure 4.9 Exemple : Partition Quadtree (Image de Lena)

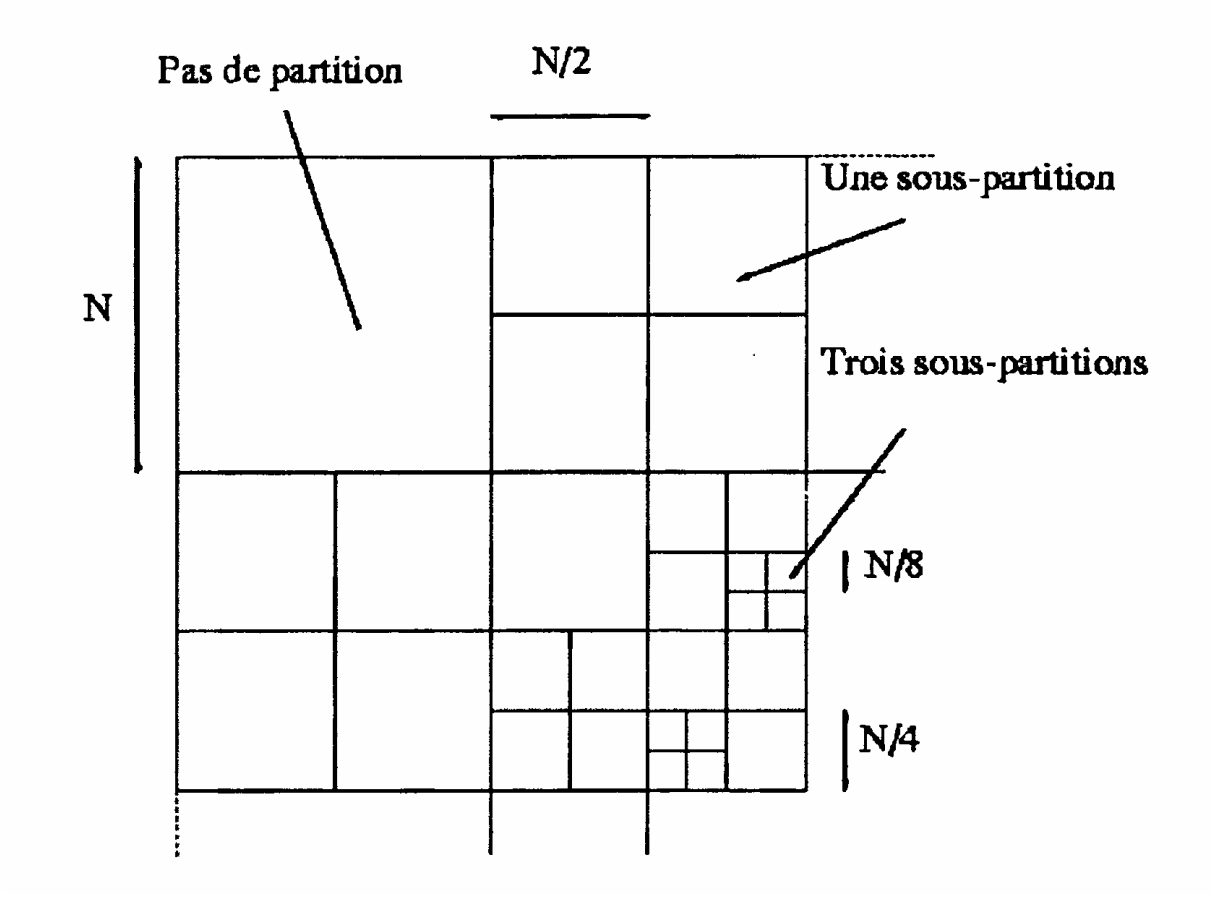

Figure 4.10 Réalisation d'une partition Quadtree

En se basant sur les résultats obtenus (Image partitionnée Figure 4.9) on remarque que : [16]

- le partitionnement est adapté au contenu de l'image c'est-à-dire prend la forme de la figure.
- le partitionnement QuadTree a un effet très important sur le nombre de bloc.
- dans une image, plus il y'a des détailles plus le nombre de bloc augmente et par conséquent le taux de compression diminue.

On peut remarquer aussi que :

❖ la taille des blocs est plus grande dans le cas où les parties sont homogènes ou dans le cas où ils sont couverts avec moins d'erreurs.

- ❖ dans le cas où les blocs sont de taille plus petite, on dit que le partitionnement atteint la taille minimale et on peut plus découper le bloc.

- On notera que ce sont les détails les plus complexes qui exigent une partition plus fine.

Recherche pratique des W_i :

Les W_i cherchés sont des applications affines contratcives qui envoient les D_i sur les R_i , en choisissant des D_i de taille double des R_i . En fait, la recherche des W_i consiste surtout à chercher les D_i eux-mêmes.

C'est à dire que R'_i est un agrandissement (au double) de R_i

En choisissant la distance euclidienne, minimiser $d(R_i,W_i(D_i))$ revient alors à chercher s et o pour minimiser :

$$R=\frac{1}{n}\sum_{i=1}^{n}\left(s.d_i+o-r_i\right)^2$$

Où n est le nombre de pixels de D_i .

R est alors appelé *RMS error* (Root Mean Square error). Comme R est une moyenne, on peut comparer des R provenant de calculs pour des tailles diverses de R_i et de plus $\sqrt{R}$ donne la différence moyenne entre $W_i(D_i)$ et R'_i .

Etant donné un R_i et un D_i , la fonction $R(s,o)$ est minimale lorsque les dérivées partielles par rapport à o et s s'annulent :

$$S\left(\sum_{i=1}^{n}d_i^2\right)+O\left(\sum_{i=1}^{n}d_i\right)-\left(\sum_{i=1}^{n}d_i r_i\right)=0$$

Et

$$S\left(\sum_{i=1}^{n}d_i\right)+n.o-\left(\sum_{i=1}^{n}r_i\right)=0$$

Ce qui donne :

Si $n\left(\sum_{i=1}^{n} d_i^2\right)-\left(\sum_{i=1}^{n} d_i\right)^2=0$ ***Alors***

$$\begin{cases} s=0 \\ o=\frac{1}{n}\sum_{i=1}^{n} r_i \\ R=0 \end{cases}$$

Sinon

$$\begin{cases} s=\dfrac{n\left(\sum_{i=1}^{n} d_i r_i\right)-\left(\sum_{i=1}^{n} d_i\right)\left(\sum_{i=1}^{n} r_i\right)}{n\left(\sum_{i=1}^{n} d_i^2\right)-\left(\sum_{i=1}^{n} d_i\right)^2} \\ o=\frac{1}{n}\left(\sum_{i=1}^{n} r_i - s\sum_{i=1}^{n} d_i\right) \\ RMS=\sqrt{\frac{1}{n}\left[\sum_{i=1}^{n} r_i^2+s\left(s\sum_{i=1}^{n} d_i^2-2\sum_{i=1}^{n} d_i r_i+2o\sum_{i=1}^{n} d_i\right)+o\left(on-2\sum_{i=1}^{n} r_i\right)\right]} \end{cases}$$

Alors le taux de compression peut être calculé avec :

$$Taux.comp=\frac{8\times(taille\ d'un\ bloc\ en\ pixels)}{\#bits\ de\ s\ +\ \#bits\ de\ o\ +\ [\log_2(8\times taille\ codebook)]}$$

C'est le calcul de cette *RMS error* qui permet de déterminer si un W_i envisagé est satisfaisant.

Si on a choisi une tolérance d'erreur, pour un R_i donné, l'algorithme pratique de recherche W_i cité dans la partition QuadTree, est alors le suivant :
On envisage tous les D_i possibles de l'image, pour chacun on calcule *s*, *o* et R_i. Une fois cela fait parmi les D_i qui ont donné un *s* compris entre 0 et 1, on retient celui qui a donné l'erreur Rb la plus faible.

Si *Rb* est inférieur à la tolérance, D_i convient, et on stocke le W_i correspondant.

Sinon comme le W_i correspondant était le meilleur possible, c'est qu'aucun W_i satisfaisant ne peut être trouvé pour le R_i donné, et donc il faut le subdiviser.

4.3.1.1.1 Représentation en arbre de la phase de division

On procède de la manière suivante : prenons une image notée A^0 de taille 2^m x2^m on construit le QuadTree de haut vers le bas en divisons récursivement tout bloc $\mathbf{A^1_i}$ non homogène selon un parcours donné (critère d'homogénéité) avec A^0 le bloc constituant la région de départ, I le niveau dans la représentation pyramidale du codage en QuadTree, et *i* le numéro du sous bloc avec $i \in \{1,2,3,4\}$.de la division d'un bloc A_i^1 de taille2^{m-1}x2^{m-1} résulte donc quatre sous-bloc A_j^{i+1} avec $j \in \{1,2,3,4\}$.a chaque division quatre nouveaux blocs sont recalculés, et ainsi de suite jusqu'à un bloc homogène. [14], [21]

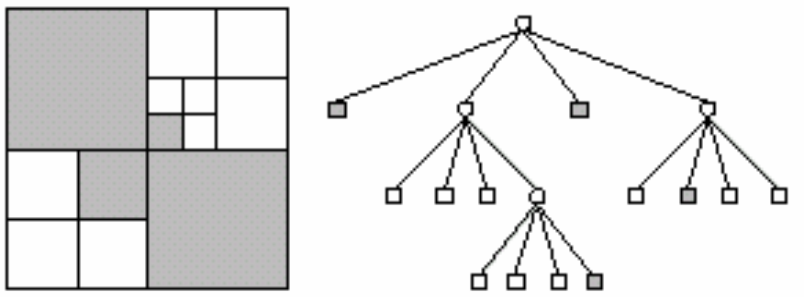

Figure 4.11 A gauche : principe de la division récursive d'une image .A droite représentation Arborescente du quadtree calculé sur une image de taille 8 × 8 pixels

Ce type de partitionnement est certainement le plus répandu dans les programmes de compression fractale. Il est simple à utiliser et fournit des résultats corrects.

4.3.2 Partitionnement semi-rigide

4.3.2.1 Partitionnement Horizontal - Vertical (H-V)

Un autre type de partitionnement, développé par Fisher et Menlove, est le partitionnement H-V.

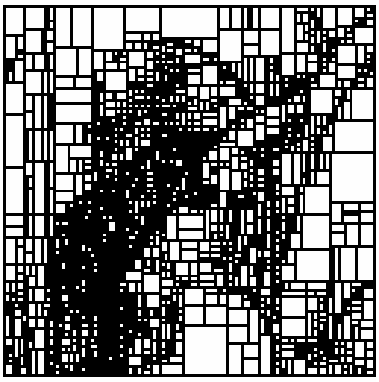

Figure 4.12 Partitionnement (H-V) calculé sur l'image de Lena et composé de 2910 rectangles

Un processus récursif de subdivision des blocs en deux sous-rectangles conduit à une partition adaptée au contenu de l'image. La partition n'est pas rigide comme le QuadTree puisque la division d'un bloc, qui tient compte de sa texture, ne crée pas nécessairement deux blocs d'égales surfaces. La partition est semi-rigide dans le sens où les arêtes des blocs demeurent obligatoirement soit horizontales, soit verticales [21], [14].

4.3.2.1.1 L'intérêt du Partitionnement H -V pour la compression par Fractales

La méthode est similaire à celle de construction des Quadtree car il y a subdivision verticale ou horizontale lorsque le bloc ne vérifier pas le critère d'homogénéité (Voire Figure 4.12). Deux différentes méthodes sont proposées par Fisher selon l'homogénéité du bloc.

1. Si le bloc est parcouru par une frontière oblique de l'image, la séparation est choisie de manière à ce que l'un des sous blocs soit parcouru diagonalement par la frontière et que s'il existe un autre sous-bloc, celui-ci soit homogène.
2. Si le bloc inclut une frontière horizontale ou verticale, la séparation est choisie de manière à créer deux sous-blocs homogènes.

- Les blocs source sont cherchés dans toute l'image.

- Lors des comparaisons entre les blocs, on considère ou non les rotations du rectangle source. Si on les prend en compte, alors le nombre de blocs sources disponibles est multiplié par quatre (car il y a 4 rotations possibles).
- Le facteur d'échelle de la transformation affine peut être contraint à n'être que positif.
- Différents facteurs de taille peuvent être ou non testés sur les hauteurs et largeurs des blocs.

Ce partitionnement présente ainsi l'avantage de contenir moins de blocs que le QuadTree car ils sont plus adaptatifs et permis ainsi d'être plus souple.

4.3.3 Partitionnement souple

L'intérêt de ces partitionnements est qu'ils sont souples puisqu'ils sont calculés sur un ensemble de points pouvant être positionnés à peu près n'importe où sur le support de l'image [14].

4.3.3.1 Partitionnement triangulaire de Delaunay

Un triangle $T=(p_i, p_j, p_K)$ où p_i, p_j, p_K sont dans P (avec $P=\{p_i \in \Re^2, i=1,...,n\}$ les n points sont appelés germes ou sites) est un triangle de Delaunay Si seulement si, l'intérieure du cercle circonscrit à T ne contient aucun point de $B(p_i, p_j, p_K) \cap (P - p_i - p_j - p_K) = \theta$ est qu'il soit borné.

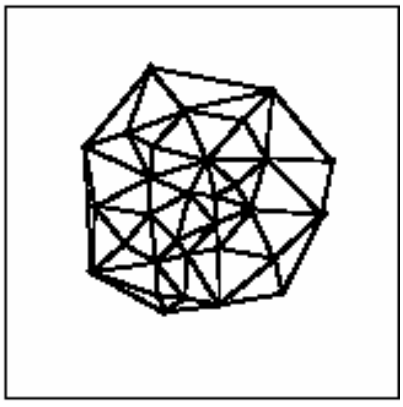

Figure 4.14 Partition de Delaunav d'un ensemble de points de P.

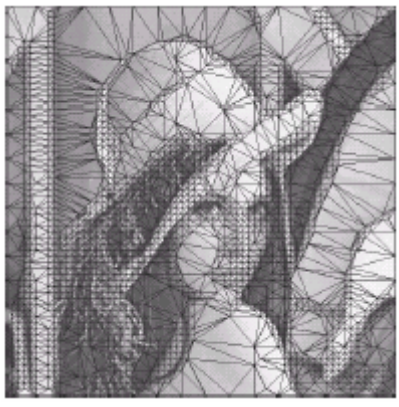

Figure 4.15 Partition triangulaire de l'image de Lena

4.4 Méthode de A. Jacquin

L'approche de A. Jacquin est fondée sur une partition *R* régulière à géométrie carrée. L'image est partitionnée en blocs destination (blocs parents) carrés de taille fixe égale à *B*B* pixels (*B* = 8). L'algorithme recherche, pour chacun des blocs destination R_i le bloc source D_i ; de taille D*D (D = 2B) qui minimise l'erreur $d(R_i,\hat{R}_i)$ ou $\hat{R}_i$ est l'approximation de R_i calculée à partir du bloc source D_i . La mesure d'erreur d est donnée par : [14]

$$d(R_i,\hat{R}_i)=\sum_{j=1}^{B^2}(R_{ij}-\hat{R}_{ij})^2$$

Où R_{ij} et $\hat{R}_{ij}$ sont respectivement les valeurs des pixels d'indice *j à* l'intérieur du bloc original R_i et du bloc collé $\hat{R}_i$. L'opération de collage, appelée collage parent.

4.4.1 Collage d'un bloc source sur un bloc destination

L'opération de collage d'un bloc source D_i sur un bloc destination R_i, réalisée par la transformation ω_i, se décompose en deux parties :

- Une transformation spatiale déforme le support du bloc D_i .
- Une transformation massique agit sur la luminance des pixels du bloc D_i déformé.

Ces deux points sont détaillés dans la suite de ce paragraphe.

- **La transformation spatiale (géométrique)**

Son rôle consiste à transformer une partie de l'image, le domaine D_i de taille *D* vers une partie de l'image, le range R_i de taille *B* avec un facteur de contraction $S = D/B$ [15].

Exemple :

Prenons le cas où *D = 2B*, les valeurs des pixels de l'image contractée sur le bloc R_i sont la moyenne de quatre pixels du bloc D_i (Voir Figure 4.4).

- **La transformation massique**

La transformation massique agit sur le bloc $b_2^{(i)}$ pour approximer le bloc destination R_i .

La complexité de cette transformation dépend de la nature du bloc R_i considéré. Jacquin propose pour cela de classifier les blocs carrés à l'aide de la méthode développée par Ramamurthi et Gersho, trois classes regroupent les blocs homogènes, les blocs texturés et les blocs avec contours (simples et divisés) de l'image [14].

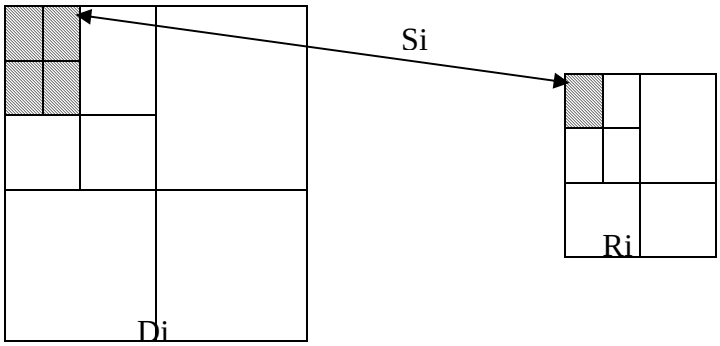

Figure 4.4 Illustration de transformation géométrique S_i

Selon la classe à laquelle appartient le bloc destination R_i, une transformation massique plus ou moins complexe lui est associée. Celle-ci dépend du bloc décimé $b_2^{(i)}$ et/ou d'un bloc constant $b_1^{(i)}$ formé de pixels tous égaux à 1. Au bloc $b_2^{(i)}$ sera associé un coefficient d'échelle noté $\beta_2^{(i)}$ et au bloc $b_1^{(i)}$ un coefficient de décalage noté $\beta_1^{(i)}$.

Le choix du type de transformation est fonction de la procédure suivante : [14]

Si ***le bloc*** R_i ***est homogène :*** absorption des niveaux de gris de R_i. Aucune recherche de blocs source D_i n'est effectuée. La transformation de R_i codée sur s bits est donnée par :

$$R_i = \beta_1^{(i)} b_1^{(i)}$$

Où l'entier $\beta_1^{(i)}$ est copris entre 0 et 255.

Si le bloc R_i ***est texturé :*** recherche des blocs sources D_i, puis modification de contraste et décalage . La transformation de D_i , codée sur m bits , est donnée par :

$$\hat{R}_i = \beta_2^{(i)} b_2^{(i)} + \beta_1^{(i)} b_1^{(i)}$$

Où $\beta_2^{(i)}$ appartient à l'ensemble $\{0.7,0.8,0.9,1.0\}$et l'entier $\beta_1^{(i)}$ est compris entre –255 et 255.

Si le bloc R_i ***contient des contours :*** recherche du bloc source D_i puis modification de contraste, décalage et isométrie ISO_i . La transformation de D_i codée sur e bits, est donnée par :

$$\hat{R}_i = ISO_i\left(\beta_2^{(i)} b_2^{(i)} + \beta_1^{(i)} b_1^{(i)}\right)$$

Où $\beta_2^{(i)}$ appartient à l'ensemble $\{0.5,0.6,0.7,0.8,0.9,1.0\}$et $\beta_1^{(i)}$ est compris entre -255 et 255.

Lorsque le bloc destination est texturé ou recouvre des contours, le coefficient d'échelle $\beta_2^{(i)}$est calculé de manière à rendre égaux les écarts types des deux blocs $b_2^{(i)}$ et R_i. Il est ensuite approximé par le coefficient appartenant à un ensemble de valeurs prédéfinies réelles positives, et inférieures à 1. Le coefficient de décalage β_1 est calculé de manière à ce que les moyennes des pixels des deux blocs $b_2^{(i)}$et R_i soient égales. Il n'est pas quantifié.

- Les transformations massiques utilisées appartiennent à deux classes :

  1ere classe :

  Les transformations de cette classe agissent seulement sur les valeurs des pixels (pour plus de détail voir [15]).

2eme Classe :

Les différentes transformations de cette classe ne font que déplacer les pixels à l'intérieur même des blocs. Elles sont appelées isométries (pour plus de détail voir [21], [15]).

- **La construction du dictionnaire**

La construction du dictionnaire est effectuée en déplaçant sur le support de l'image un bloc carré d'un pas de 4 pixels dans les directions horizontales et verticales. Lorsque deux blocs sont comparés, les huit isométries discrètes sont considérées. Pour une image de taille 256 * 256, une telle recherche est ainsi effectuée au travers d'une librairie composée de Q blocs source où :

$$Q = 8\left(\frac{256-16}{4}+1\right)^2 = 29768$$

Dans le cas d'une image de taille 512 * 512 pixels, le nombre Q de blocs source s'élève à 125000 [14].

4.4.2 Collage enfant

Jacquin propose dans un deuxième temps de rediviser les blocs parents collés $\hat{R}_i$ en quatre sous-blocs destination de taille 4 * 4 pixels appelés ***blocs enfants*** (voir figure 4.5). Les blocs obtenus sont comparés à leurs correspondants dans l'image originale, avec B = 4. Si l'erreur est supérieure à un seuil donné, ils sont codés séparément en recherchant dans l'image le meilleur bloc source de taille 8*8. Le processus de collage est dans ce cas appelé *collage enfant***.** Si, pour un bloc parent, trois ou quatre collages enfant sont nécessaires, seuls sont codés les 4 collages enfant. Si un ou deux collages enfant sont nécessaires, le bloc parent est codé par le collage parent complété des collages enfants [14].

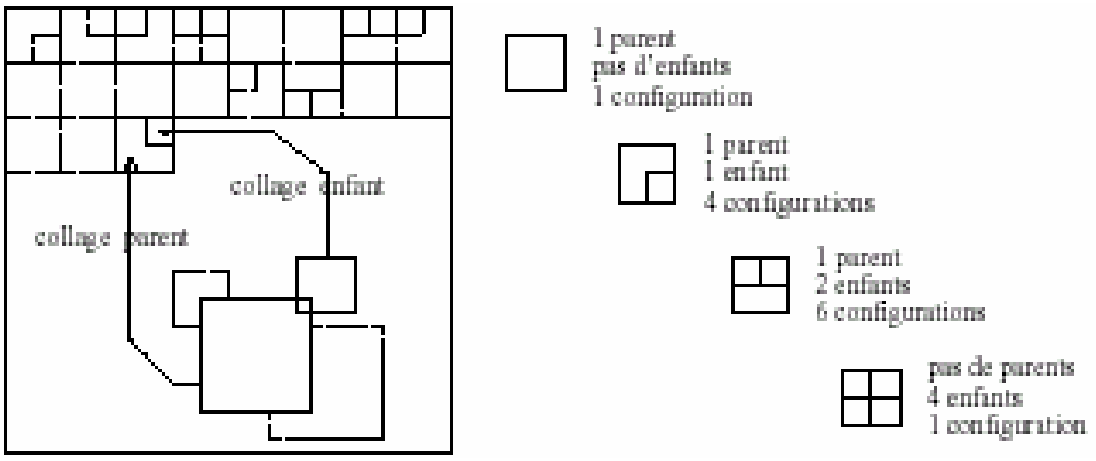

Figure 4.5 Partitionnement formé de blocs parents et enfants

- **Mémorisation des coefficients de collage**

La mémorisation du collage d'un bloc source (parent ou enfant) D_i sur un bloc destination (parent ou enfant) R_i comprend :

- ✓ L'indice du bloc source D_i retenu parmi les Q blocs de la librairie à condition que ceux-ci soient rangés dans une liste de blocs et que leur organisation sur le support de l'image soit connue. Sinon, il est nécessaire de mémoriser les coordonnées (x, y) d'un pixel de référence dans le bloc D_i (par exemple le coin supérieur gauche dans le cas d'un bloc carré).
- ✓ L'isométrie utilisée lors du collage (une parmi huit).
- ✓ Les coefficients β_1 et β_2 de la transformation massique.

Cette information est associée à chacun des N blocs destination de la partition R

4.5 Méthode de Y.Fisher

Y. Fisher décrit l'opération de collage d'un bloc source sur un bloc destination en utilisant une formule unique donnée par :

$$W_n\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_n & b_n & 0 \\ c_n & d_n & 0 \\ 0 & 0 & \beta_2^{(n)} \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e_n \\ f_n \\ \beta_1^{(n)} \end{pmatrix}$$

Où (x, y) sont les coordonnées d'un pixel intérieur au bloc source $d_{\alpha(n)}$ et *z* est le niveau de gris du pixel. a_n, b_n, c_n, d_n, e_n et f_n, sont les coefficients de la transformation spatiale affine ramenant les pixels du bloc source $d_{\alpha(n)}$ à l'intérieur du bloc destination R_i.

$\beta_1^{(n)}$ et $\beta_2^{(n)}$ sont les coefficients de la transformation du niveau de gris des pixels. [14]

Alors que les coefficients de la transformation massique utilisée par Jacquin sont choisis dans des ensembles prédéfinis de valeurs, Y. Fisher utilise un quantificateur scalaire uniforme. Jacobs et al. ont montré que pour ce type de quantificateur, la quantification des coefficients de translation et d'échelle, respectivement sur 7 et 5 bits, est optimale en terme de qualité visuelle des images reconstruites. [14]

4.6 Méthode de F. Dudbridge

Dudbridge a proposé en 1992 une méthode rapide de compression des images par fractales basée sur un partitionnement carré régulier.

La rapidité de l'algorithme de compression est due au fait qu'aucune recherche de similarité inter-blocs n'est faite [14].

L'image est partitionnée en un ensemble de blocs carrés, de taille fixe, puis chacun des blocs est codé individuellement par une transformation fractale.

La méthode donne de moins bons résultats que par exemple la méthode de Jacquin.

4.7 Synthèse comparative des méthodes

- **Fractale & JPEG** [15]

Les images utilisées sont de tailles 256*256 pour la plupart, mais aussi d'autres tailles. Le codage des valeurs s'est fait comme suit :

5 bits pour le contraste.

7 bits pour la luminosité.

1 bit pour stocker la position du bloc rang.

3 bits pour la transformation.

Le tableau 4.1 montre des résultats de Compression / Décompression par fractale obtenu sur quelques images.

image	*Taille image*	*Temps de compression*	*Taille après compression*	*Gain fractale*	*PSNR (db)*	*Nombre de blocs rangs*
Lena 1	66KO 256x256x8	13 S	9 KO	7.20	30.10	2935
Collie	66KO 256x256x8	12 S	10 KO	6.92	33.77	3046
Cameramen	66KO 256x256x8	11 S	7 KO	10.14	27.96	2110
Saturn	119KO 400x300x8	14 S	6 KO	21.48	38.74	1944
Image	66KO 256x256x8	6 S	4 KO	17.04	35.55	1282
Lena 2	258KO 512x512x8	1MIN.45 S	25 KO	10.30	33.90	7639
Humbird	193KO 256x256x24	34 S	23 KO	8.51	R:28.04 G:28.24 B:27.75	R:24.25 G:28.98 B:28.12
Portrait	352KO 300x400x24	1MIN.36 S	42 KO	8.41	R:27.91 G:28.00 B :29.82	R:44.88 G:42.60 B :45.42
Pupup	193KO 256x256x24	49 S	9 KO	8.08	R :28.63 G :28.55 B :30.01	R :29.92 G:27.31 B:28.03

Tableau 4.1 Résultats obtenus avec la méthode Fractale

Le tableau 4.2 montre des résultats de compression / décompression par JPEG obtenus sur les même images.

image	*Taille image*	*Temps de compression*	*Taille après compression*	*Gain JPG*	*PSNR (db)*
Lena 1	66KO 256x256x8	1 S	13 KO	5.07	36.67
Collie	66KO 256x256x8	1 S	11 KO	6	42.11
Cameramen	66KO 256x256x8	1 S	12 KO	5.50	40.13
Saturn	119KO 400x300x8	2 S	9 KO	13.22	24.62
Image	66KO 256x256x8	1 S	7 KO	9.42	35.55
Lena 2	258KO 512x512x8	2 S	34 KO	7.58	39.25
Humbird	193KO 256x256x24	3 S	57 KO	3.38	R:30.10 G:31.51 B:29.20
Portrait	352KO 300x400x24	3 S	113 KO	3.11	R:29.54 G:30.01 B :31.45
Pupup	193KO 256x256x24	3 S	67 KO	2.88	R :30.04 G :30.47 B :32.08

Tableau 4.2 Résultats obtenus avec la méthode JPEG

- **Fractale & JPEG & Ondelette**

La figure 4.16 montre les résultats obtenus après les différents types de compression:

Le résultat comparatif s'agit d'illustrer les critères principaux concernant la compression, telle la vitesse moyenne de compression, le taux de compression moyen (ratio) et la fidélité moyenne (erreur RMSE) qu'induisent ces procédés.

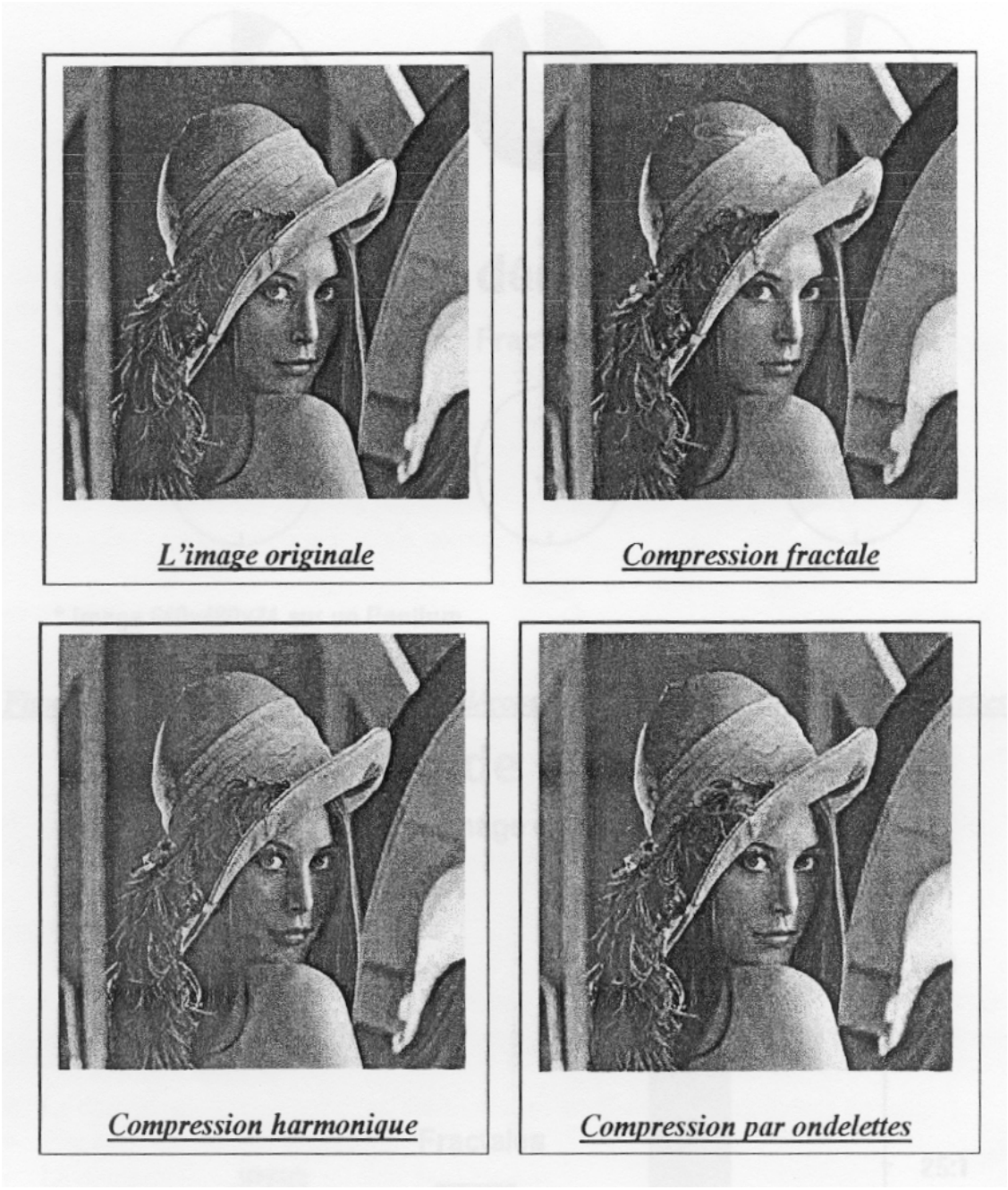

Figure 4.16 Les résultats obtenus après les différents types de compression

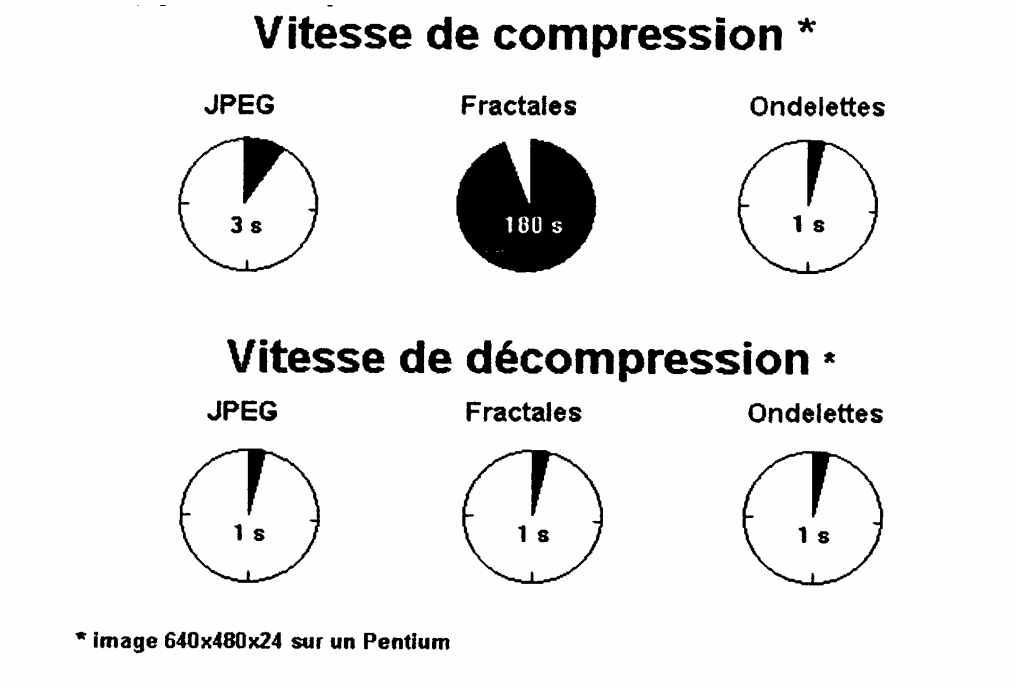

Figure 4.17 Vitesse de compression & décompression « JPEG, Fractales, Ondelettes»

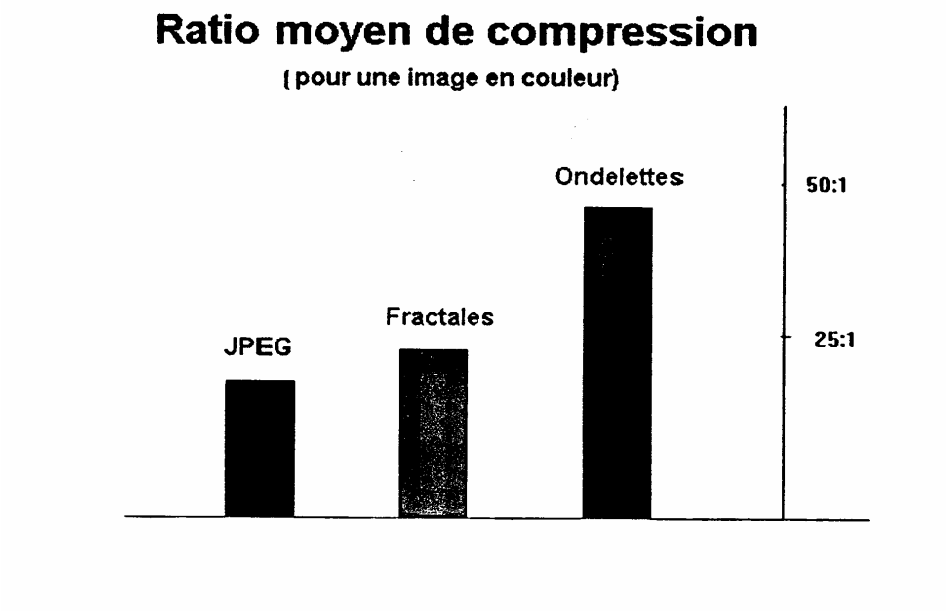

Figure 4.18 Ratio moyen de compression « JPEG, Fractales, Ondelettes

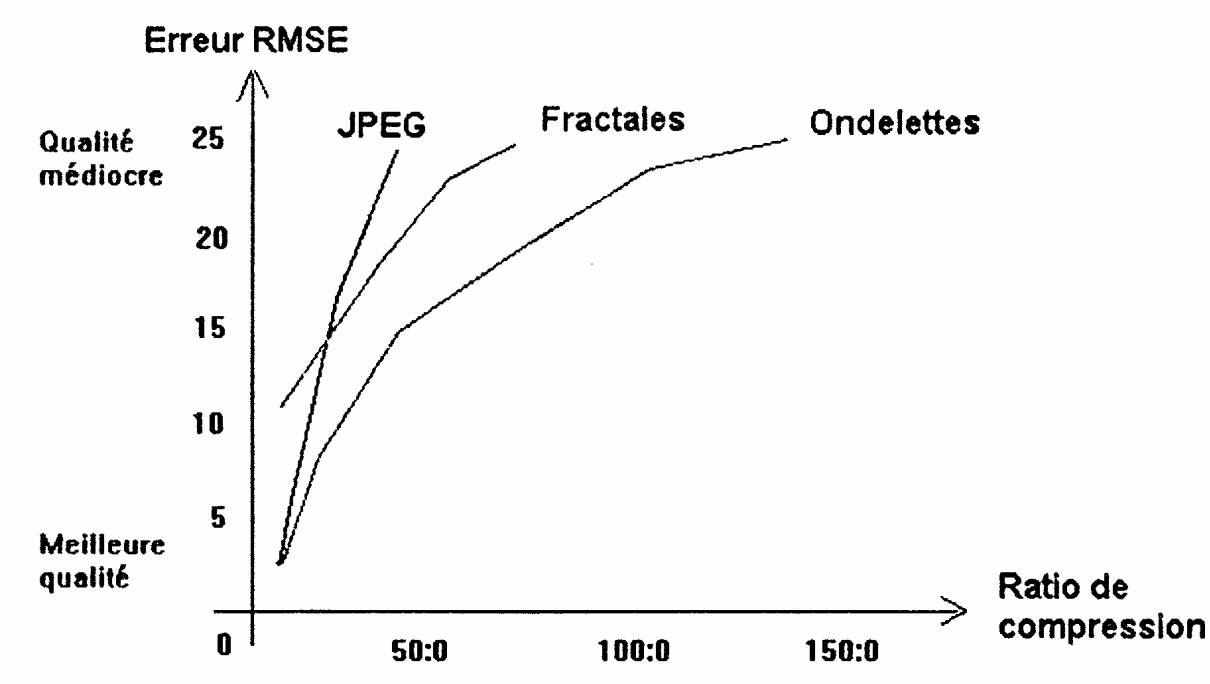

Figure 4.19 Erreur RMSE « JPEG, Fractale. Ondelettes »

D'après les schémas et les graphes présentés ci-dessus, nous pouvons conclure les avantages et les inconvénients de la méthode fractale comme suivent :

Les avantages :

- La compression fractale permet un puissant zoom fractal pour agrandir une image sans effet de pixellisation. (ce mode de compression est donc recommandé pour la compression de photographies numériques, avec des ratios de compression ne dépassant pas 20:1 ou bien dans les applications multimédia et cd-rom...).
- Son taux de compression est très avantageux, tout en conservant un aperçu fidèle.
- Un effet de lissage flou disponible en traitement d'images.

Les inconvénients :

- Ce type de compression n'a pas encore fait l'objet de l'édition d'une norme : son utilisation n'est donc pas encore totalement démocratisée. Ce mode est donc moins fiable que le mode harmonique.
- La compression reste lente malgré toutes les améliorations (coûteuse en temps).

- Elle peut nécessiter un hardware spécialisé.
- La compression produit un flou dans l'image.
- Ce mode de compression est à éviter pour les images de poids supérieur à 2 Mo.

4.8 Conclusion

Après avoir étudié la méthode Fractale, nous pouvons estimer que cette dernière est une méthode de compression efficace qui nous donne des taux très élevés et qui nous permet de gagner de l'espace de stockage, réduire la taille de l'image et minimiser le temps de transmission.

Les chapitres suivants sont consacrés au cœur de notre travail qu'est l'implémentation d'un environnement parallèle pour la compression d'images à l'aide des fractales.

Chapitre 5

Parallélisme et architectures parallèles

L'architecture des ordinateurs, qu'il s'agisse de microprocesseurs ou de supercalculateurs, est fortement influencée par l'exploitation d'une propriété fondamentale des applications : le parallélisme. Un grand nombre d'architectures présentes dans les sites informatiques sont parallèles. Ce type d'architecture touche une large gamme de machines depuis les PC bi-processeurs jusqu'aux supercalculateurs. Aujourd'hui, la plupart des serveurs sont des machines parallèles (des multiprocesseurs)[34].

Dans ce chapitre nous allons introduire la notion de parallélisme, de discuter la nécessité de l'exploiter pour atteindre de hautes performances et de présenter les principales architectures parallèles spécialisées pour le traitement d'images et en particulier la compression d'images à l'aide des fractales.

5.1 Motivation pour le parallélisme [34]

L'exploitation du parallélisme dans l'architecture des ordinateurs est née de la conjonction de trois éléments : les besoins des applications, les limites des architectures séquentielles et l'existence dans les applications de la propriété de parallélisme.

5.1.1 Besoins des applications

La notion de parallélisme est souvent attachée à celle de la performance d'exécution des applications. Ce dernier terme recouvre différentes notions suivant les besoins des applications. En effet, quelque soit le domaine d'application, le parallélisme peut être exploité pour répondre à deux besoins : la puissance de traitement et/ou la disponibilité.

La puissance de traitement recouvre deux grandes notions : la latence de traitement et le débit de traitement. La latence représente le temps nécessaire pour l'exécution d'un traitement. Le débit représente le nombre de traitements exécutables par unité de temps.

Ces deux notions peuvent être indépendantes. Réduire la latence est plus difficile qu'augmenter le débit. Dans le premier cas, il s'agit de lutter contre le

temps qui en dernier ressort est fixé par les possibilités technologiques. Alors que, dans le deuxième cas, si plusieurs traitements sont indépendants, l'augmentation du nombre de ressources suffit pour exécuter plus de traitements en même temps.

La puissance de traitement dépend aussi de la capacité et de l'organisation de la mémoire d'un ordinateur. Certaines applications requièrent des ensembles de données dont la taille est supérieure à la capacité d'adressage d'un ordinateur séquentiel. Multiplier les ressources qui adressent la mémoire permet d'accroître la taille de la mémoire adressable. Certaines organisations d'architectures parallèles permettent donc d'adresser plus de mémoire que des architectures séquentielles.

La majorité des applications requérant de hautes performances appartiennent au « supercomputing » ou au « commercial computing ». Le premier domaine concerne les applications du traitement numérique alors que le deuxième concerne principalement les applications de bases de données. Ces deux domaines recouvrent principalement quatre types d'applications : la simulation numérique ou entière, l'analyse extensive de grands volumes d'informations, les serveurs de ressources, les applications à contraintes (temps réel, …).

Pour certaines applications, la disponibilité des données (base de données) ou la capacité de réaliser un traitement (applications spatiales) est essentielle et doit rester permanente. La disponibilité est fortement liée à la capacité de résister à des pannes logicielles et matérielles. Dans la résistance aux pannes, l'objectif premier consiste à traduire l'occurrence d'une panne par une diminution la plus réduite possible des performances (temps de traitement plus long ou débit réduit). Un objectif plus difficile est d'assurer une diminution progressive des performances en fonction du nombre de pannes.

La résistance aux pannes suppose que l'arrêt d'une partie de l'ordinateur (pour panne, maintenance, etc.) n'entraîne pas son arrêt complet. Si l'élément arrêté est le processeur, il est évident que seule une redondance permet de résister à un manque. Un mécanisme de surveillance est aussi nécessaire pour détecter l'occurrence de pannes. Pour assurer le fonctionnement en cas de manque, le traitement sur l'élément arrêté est transféré sur l'un des éléments redondants. La redondance repose donc sur la mise en parallèle d'éléments assurant les mêmes fonctionnalités.

5.1.2 Limites de l'approche microprocesseur

L'utilisation simultanée de plusieurs ressources entraîne la nécessité de gérer la coordination de ces ressources et complexifie la programmation.

Aussi, l'exploitation du parallélisme ne peut se concevoir qu'en réponse aux limitations de l'exécution séquentielle. Les ordinateurs séquentiels (équipés d'un seul processeur) sont construits autour des microprocesseurs. La standardisation de ce composant fait que les caractéristiques des différents microprocesseurs sont très semblables. Les limites de l'exécution séquentielle découlent donc des limites des microprocesseurs actuels : performance, capacité d'accès à la mémoire, tolérance aux pannes, variétés des formats de données et des opérateurs.

Le parallélisme permet de palier le problème de la limite de capacité d'accès d'un microprocesseur à la mémoire en multipliant les bancs mémoire et en rajoutant des mécanismes matériels d'extension d'adressage. Dans ce cas, chaque accès mémoire nécessite plusieurs instructions.

La principale motivation est la limite en performance des microprocesseurs. L'objectif de 1 TeraFlops avant l'an 2000 ne peut être atteint par un microprocesseur même si l'évolution des performances des microprocesseurs depuis 1985 suit une courbe exponentielle dans le temps. L'IBM Power III [35] annoncé pour 1999 était l'un des microprocesseurs les plus performants en l'an 2000. Sa performance est de 2 GigaFlops ($2*10^9$ *Flops*) avec une fréquence de 500 Mhz. Il faudra 500 Power III pour approcher de 1 TeraFlops.

Les microprocesseurs possèdent aussi d'autres limites comme l'inadéquation de leur format de données et de leurs opérations aux caractéristiques de certaines applications telles que le traitement d'images ou la comparaison de séquences biologiques. Il peut être préférable, en termes de performance, de coût et de compacité, d'avoir recourt à une réalisation parallèle spécifique, c'est-à-dire de concevoir un ordinateur dont les opérations de traitement, l'organisation mémoire, etc. sont adaptés aux calculs à réaliser.

5.2 Définition du parallélisme

C'est le parallélisme présent dans une application qui permet d'exécuter simultanément, par des ressources matérielles différentes, plusieurs parties de cette application. Pour introduire la notion de parallélisme nous allons examiner la boucle suivante :

Pour i de 1 à n faire

A[i] ← B[i] + C[i]

FinPour

Le corps de cette boucle ne comporte qu'une seule opération. Si cette boucle présente du parallélisme, il faut le chercher entre les itérations. Voici les trois itérations :

A[1] ← B[1] + C[1] (a)

A[2] ← B[2] + C[2] (b)

A[3] ← B[3] + C[3] (c)

La sémantique introduite par le programmeur indique les résultants attendus en mémoire après l'exécution de la boucle. Quelque soit l'ordre d'exécution ex : a, b, c ou c, b, a ou b, c, a etc. les résultats en mémoire sont identiques. La sémantique du programme ne dépend pas de l'ordre d'exécution de cette boucle. En particulier, l'exécution simultanée de ces trois itérations respecte la sémantique. Les itérations de cette boucle peuvent donc être exécutées en parallèle.

5.2.2 Définition formelle

Pour étudier la présence de parallélisme dans une application, nous avons besoin d'outils plus formels.

Bernstein [36] a introduit en 1966 un ensemble de conditions permettant d'établir la possibilité d'exécuter plusieurs programmes (processus) en parallèle. Supposons deux programmes : P1 et P2. Supposons que chacun utilise des variables en entrée et produise des résultats en sortie. Nous parlerons des variables d'entrée de P1 et P2 (respectivement E1 et E2) et des variables de sortie de P1 et P2 (respectivement S1 et S2).

Selon Bernstein, les programmes P1 et P2 sont exécutables en parallèle (notation P1 || P2) si les conditions suivantes sont respectées :

$$\{E1 \cap S2 = \phi, E2 \cap S1 = \phi, S2 \cap S1 = \phi\}$$

plus généralement, un ensemble de programmes P1, P2 … Pk peuvent être exécutés en parallèle ssi les conditions de Bernstein sont satisfaites c'est-à-dire si Pi || Pj pour tout couple (i, j) avec $i \neq j$.

L'exemple intuitif et les conditions de Bernstein introduisent la notion de dépendances entre deux ou plusieurs programmes (ou opérations). Pour que deux programmes ou deux opérations puissent être exécutées en parallèle, il faut :

- 1) qu'ils (elles) soient indépendants (elles),
- 2) que l'on puisse détecter cette indépendance et
- 3) qu'il existe suffisamment de ressources pour les exécuter simultanément.

5.3 Les sources du parallélisme

Pour découvrir d'où vient le parallélisme qui est exploité lors de l'exécution d'un programme, il faut remonter jusqu'au problème que traite le programme. Le problème doit présenter intrinsèquement des parties parallèles. Mais entre le problème dans sa formulation abstraite et le programme qui s'exécute, il existe plusieurs étapes : la formulation d'un algorithme, la programmation et la compilation. Le but, bien entendu, est de préserver autant que possible le parallélisme lors de la traversée des étapes.

Voici quelques une des applications typiques du parallélisme : calcul numérique, traitement et synthèse d'images, traitement du signal, base de données, systèmes experts, systèmes client-serveur, etc. les algorithmes utilisés dans ces domaines travaillent sur des structures de données de grandes tailles (ex : des tableaux de plusieurs millions d'éléments) et/ou effectuent des actions indépendantes. Il existe donc deux formes de parallélisme :

- une issue des données : le parallélisme de données et,
- l'autre issue de la structure de l'application : le parallélisme de contrôle.

5.3.1 Le parallélisme de données

C'est un type de parallélisme pour lequel la même opération est réalisée simultanément par de nombreux processeurs sur des données différentes [37]. Cette définition recouvre deux notions : la présence d'un parallélisme issu des données et la manière d'exploiter ce parallélisme.

Prenons un exemple simple d'algèbre linéaire et plus particulièrement de calcul matriciel. L'addition de deux matrices consiste, pour tout les éléments de mêmes indices des deux matrices opérandes à les additionner et à stocker le résultat dans l'élément de mêmes indices de la matrice résultat. Voici la boucle correspondante pour les matrices opérandes B et C et la matrice résultat A :

```
Pour i de 1 à n
  Pour j de 1 à n
    A[ i ][ j ] ← B[ i ][ j ] + C[ I ][ j ]
  FinPour
FinPour
```

Comme pour la boucle étudiée en 5.2, les itérations de cette boucle sont indépendantes. Il y a n^2 itérations avec une opération par itération. Le potentiel de parallélisme exploitable dans cette boucle est donc n^2 opérations simultanées. L'ampleur du potentiel de parallélisme exploitable dépend directement de la taille des structures de données manipulées.

L'exploitation de ce parallélisme est fondamentale car les structures de données manipulées dans les applications numériques, les traitements de base de données, le traitement du signal et de l'image sont généralement très grandes : matrices de plusieurs milliers d'éléments de côté, base de données avec des millions d'entrées, plus d'un million de pixels par image. Les données sont de loin la source de parallélisme qui offre le plus de potentiel.

La définition indique aussi une manière d'exploiter ce parallélisme. Il s'agit d'utiliser de nombreux processeurs simultanément et de leur faire exécuter la même opération. Généralement le nombre de processeurs est beaucoup plus petit que le parallélisme potentiel et chaque processeur devra donc traiter plusieurs données.

5.3.2 Le parallélisme de contrôle

C'est un type de parallélisme pour lequel des opérations différentes sont réalisées simultanément [37]. Ce parallélisme peut provenir de l'existence dans le programme de fonctions indépendantes. Il peut aussi provenir d'opérations indépendantes dans une suite d'opérations. Ce parallélisme ne dépend donc pas des données mais de la structure du programme à exécuter. C'est l'absence de

dépendances entre différentes parties du programme (quelque soit leur taille : fonctions, boucles, opérations) qui est la source du parallélisme de contrôle.

Voici un exemple très simple de programme pour un serveur :

```
faire toujours
    détecter demande-client
    si demande client vrai
        lancer traitement-client
    finsi
finfaire
```

voici le programme correspondant pour le traitement client :

```
début
  ouvrir fichier-client
  faire traitement-demande
  fermer fichier-client
fin
```

Supposons que la fonction *lancer* retourne immédiatement après avoir lancé le programme traitement-client ; c'est-à-dire sans attendre que celui-ci se termine. Dans ce cas, le programme du serveur continue à exécuter alors que le programme traitement-client est en cours d'exécution. Si il y a suffisamment de ressources, ces deux programmes seront exécutés simultanément.

Si le serveur reçoit une nouvelle demande-client, il lancera l'exécution du nouveau traitement-client de la même manière. Il est donc possible, à un instant donné, que plusieurs traitement-client s'exécutent en même temps (avec des niveaux d'avancement différents) en plus du programme serveur. Comme ces programmes peuvent réaliser des opérations différentes, il ne s'agit pas de parallélisme de données mais bien de parallélisme de contrôle.

L'exploitation de ce parallélisme suppose que les processeurs fonctionnant simultanément soient capables de dérouler leur propre programme (puisque les traitements peuvent être tous différents).

5.4 Les architectures parallèles

Une architecture parallèle est le résultat d'un équilibre entre plusieurs paramètres : la nature des ressources, les performances relatives de ces ressources, l'agencement des ressources et leur mode de contrôle.

Les architectures parallèles sont généralement construites à partir des ressources qui composent les architectures séquentielles : unité de traitement, unité de contrôle (séquencement du programme), mémoire, entrées sorties (disque, réseau, etc.). Durant l'exécution d'un traitement parallèle, toutes les unités échangent des informations. Ces transferts d'informations sont réalisés par une ressource supplémentaire : le réseau de communication interne.

5.4.1 Classification des architectures parallèles

La diversité des ressources et leur nombre important conduit à une grande variété d'agencements possibles de ces ressources dans une architecture parallèle. Pour distinguer des familles d'architectures, on utilise des classifications.

La classification la plus utilisée est connue historiquement sous le nom de classification de Flynn. Elle classe les architectures suivant les relations entre les unités de traitement et les unités de contrôle. Les unités de traitement calculent chacune un flux de données « Data stream ». Les unités de contrôle déroulent chacune un programme (un flux d'instructions ou 'Instruction stream'). Le tableau suivant présente cette classification.

	1 flux d'instructions	> 1 flux d'instructions
1 flux de données	---	MISD (Pipeline)
>1 flux de données	SIMD	MIMD

Classification des architectures parallèles selon Flynn

Dans une architecture SIMD (Single Instruction stream Multile Data stream), une seule unité de contrôle gère le séquencement du programme pour plusieurs unités de traitement. En pratique, les unités de traitement fonctionnent en synchronisme et reçoivent les mêmes instructions en même temps. Comme chaque unité de traitement calcule sur un flux de données différent (des données différentes), la même opération est appliquée à plusieurs données simultanément.

Dans les architectures MISD (Multiple Instruction stream Single Data stream), un seul flux de données reçoit plusieurs traitements simultanément. Le mode pipeline d'exploitation du parallélisme de contrôle correspond assez bien à cette classe.

L'organisation MIMD (Multiple Instruction stream Multiple Data stream) est la plus utilisée. Elle découle de l'architecture des microprocesseurs actuels. Dans ces microprocesseurs, on trouve une unité de contrôle et une ou plusieurs unités de traitement. En répliquant les microprocesseurs, on réplique donc, dans la même proportion, les unités de contrôle et les unités de traitement. Lors de l'exécution d'un programme parallèle, plusieurs flux d'instructions sont donc déroulés pour traiter plusieurs flux de données.

Le modèle MIMD permet d'exécuter un programme différent sur tous les processeurs. En pratique, écrire des programmes différents est très difficile et souvent inutile. Aussi, on préfère exécuter le même programme sur tous les processeurs. Bien évidemment, le programme traite des données différentes sur chaque processeur. Ce mode d'exécution est nommé SPMD (Single Programme Multiple Data).

5.4.2 Modèles d'exécution

La réplication des ressources et leur agencement ne suffis pas à faire fonctionner une architecture parallèle. L'ensemble doit être contrôlé par un modèle d'exécution. Le modèle d'exécution est aussi une représentation intermédiaire entre le programme qu'exprime l'utilisateur dans un langage de programmation et les programmes qui vont s'exécuter sur les unités de traitement.

Il existe plusieurs modèles d'exécution dont trois ont été ou sont plus particulièrement utilisés : le modèle muti-séquentiel, le modèle dataflow et le modèle systolique. Nous ne présenterons que le modèle le plus simple et le plus utilisé. Les autres modèles sont décrits dans [38].

Le modèle multi-séquentiel correspond (comme le modèle MIMD) lui aussi à l'architecture des microprocesseurs actuels. Dans le principe, ce modèle est extrêmement simple. Il s'agit de faire fonctionner chaque microprocesseur d'une machine parallèle comme un processeur de machine séquentiel exécutant son propre programme.

Sur chaque processeur, il existe donc un compteur de programme géré localement, à partir duquel le programme du processeur est déroulé. La coopération entre les processeurs, qui vont fonctionner à leur propre rythme et de façon asynchrone, passe alors par l'échange d'informations et une gestion de leur coordination.

Le principe général suivit pour établir le programme de chaque processeur consiste, en simplifiant à l'extrême, à compiler le programme initial comme pour une exécution séquentielle puis à répartir sur les processeurs les itérations des boucles qui seront exécutées en parallèle. Chaque processeur se voit alors attribué des bornes de boucles différentes.

Des opérations de coordination sont introduites aux points du programme où des contraintes temporelles doivent être respectées.

Chapitre 6

Parallèlisation d'un algorithme de compression fractale

6.1 Introduction

Ces dernières années les techniques de compression par fractales d'images (IFS), ont gagné plus d'intérêt en raison de leur capacité de réaliser des taux de compression très élevés tout en mettant à jour la qualité très bonne de l'image reconstruite. L'inconvénient principal de telles techniques est le temps de calcul très élevé nécessaire pour déterminer le code IFS (l'image compressée).

Plusieurs techniques sont utilisées pour améliorer le temps de compression. Dans ce chapitre nous allons introduire le parallélisme dans la compression fractale d'images fixes en parallélisant l'algorithme de base non adaptatif en utilisant une architecture parallèle MIMD à mémoire distribuée pour améliorer la vitesse de compression.

6.2 Pourquoi la compression par fractales

Les méthodes standard de compression d'images peuvent être évaluées par leur taux de compression : le quotient de la mémoire occupée par l'image comme collection de pixels par la mémoire nécessaire pour stocker l'image sous forme compressée [5].

M. Barnsley suggéra que de stocker les images comme collections de transformations pouvait permettre de les compresser. La feuille qui porte son nom peut être générée par seulement quatre transformations affines. Chaque transformation affine w_i est définie par six nombres, a_i, b_i, c_i, d_i, e_i, et f_i qui ne nécessitent pas beaucoup de mémoire pour être stockés sur un ordinateur(ils peuvent être stockés dans 4 transformations x 6 nombres/ transformation x 32 bits/ nombre = 768 bits).

Stocker l'image de la feuille comme ensemble de pixels requiert beaucoup plus de mémoire (au moins 65536 bits) [5].

Comme on l'a vu, la feuille peut être générée à partir de 768 bits d'information mais nécessite 65536 bits à stocker comme collection de pixels, donnant un taux de compression de 65536/768 = 85.3 pour 1[5].

On remarque que le taux de compression d'images par fractales est élevé, ce qui justifie notre choix de cette méthode de compression.

Nous avons effectué un test sur l'image de Lena de taille 256x512.

La compression de cette image à l'aide de la méthode fractale nous a donné le taux suivant : 128Ko/22Ko = 5.81.

La compression de cette image à l'aide de la méthode Jpeg nous a donné le taux suivant : 128Ko/89.7Ko = 1.42.

Le test que nous avons fait montre l'intérêt de la compression par fractales par rapport à la méthode Jpeg qui est largement utilisée et ce justifie encore une fois le choix de la méthode de compression par fractales.

6.3 L'approche séquentielle

6.3.1 Evaluation de la complexité de l'algorithme des IFS [21]

La détermination du code fractal (IFS) d'une image fixe par l'algorithme séquentiel de la compression fractale nécessite le passage plusieurs fois par les mêmes étapes (la boucle principale de l'algorithme fractale), autrement dit, l'algorithme des IFS doit être répété plusieurs fois.

Nous allons déterminer la complexité du temps de codage de l'algorithme de compression par IFS. Les étapes de l'algorithme doivent être répétées pour toute la partition R d'une image numérique A de taille NxN. Pour chaque bloc range R_i comparé à tout les blocs domaines D_j de "la domain pool" D le nombre de répétition de comparaison sera donné par:

$$Nb_{etap} = 8 * (N/n)^2 * ((N - (2*n - 1))/p)^2 \quad , \quad \text{avec:}$$

le nombre de transformations isométriques est 8. $(N/n)^2$ est le nombre de blocs ranges R_i , p est un facteur de recouvrement entre "la domain pool" et $((N - (2*n - 1))/p)^2$ blocs domaines.

L'algorithme de compression fractale exige de calculer autant de fois la dérivée de l'expression qui donne les paramètres de la transformation affine W_i : s, o et l'erreur *rms*.

$$\begin{cases} s=\dfrac{n\left(\sum_{i=1}^{n} d_i r_i\right)-\left(\sum_{i=1}^{n} d_i\right)\left(\sum_{i=1}^{n} r_i\right)}{n\left(\sum_{i=1}^{n} d_i^2\right)-\left(\sum_{i=1}^{n} d_i\right)^2} \\ o=\dfrac{1}{n}\left(\sum_{i=1}^{n} r_i - s\sum_{i=1}^{n} d_i\right) \\ RMS=\sqrt{\dfrac{1}{n}\left[\sum_{i=1}^{n} r_i^2+s\left(s\sum_{i=1}^{n} d_i^2-2\sum_{i=1}^{n} d_i r_i+2o\sum_{i=1}^{n} d_i\right)+o\left(on-2\sum_{i=1}^{n} r_i\right)\right]} \end{cases}$$

la dernière équation comporte 13 exécutions de virgule flottante. En conséquence, l'algorithme de compression par IFS exige un certain nombre de virgules flottantes flops donné par la formule suivante:

$$N_{flops} = 8 * (N/n)^2 * ((N – (2*n – 1))/p) * (2*n^2 + 21)$$

La valeur de N_{flops} est généralement très grande ce qui rend le temps de codage trop lourd. Par exemple, si nous choisissons n=8 et p=4, la compression d'une image de taille 512x512 exige le nombre 75x10 flops c.-à-d. une demi-heure sur un processeur de 200 Mflops.

D'autre part, l'algorithme d'IFS permet l'exploitation du parallélisme massive ou le traitement massivement parallèle (MPP Massively Parallel Processing) qui semble être une réponse raisonnable au problème de codage Ifs, permettant d'augmenter la vitesse de compression et converge vers une image reconstruite de haute qualité.

6.3.2 Inconvénient de l'approche séquentielle

La compression d'images par fractales exploite les similarités au sein des images. Ces similarités sont décrites par une transformation affine contractante W de l'image dont le point fixe G est proche de l'image elle même W(G)=G. la transformation d'images consiste en des blocs de transformation qui approche les petites portions de l'image par d'autres plus larges en utilisant des transformations affines contractantes. Les petites portions sont appelées « blocs ranges R_i » et les plus larges sont appelées « blocs domaines D_i ». Tous les blocs

ranges forment une partition de l'image. Les blocs domaines peuvent être choisis librement au sein de l'image et peuvent s'imbriquer. Pour chaque range R_i un domaine D_i approprié doit être trouvé. Pour juger de la qualité d'un seul domaine, une erreur carrée moyenne (***rms***) de la distance entre le bloc range R_i et $W_i(D_i)$ est calculée et doit être inférieur à une tolérance Δ [39].

Le bloc domaine le plus similaire pour un bloc range spécifique peut être trouvé par une recherche à travers tous les domaines et leurs isométries (réflexion, rotation, …).

Dans un algorithme non adaptatif pour chaque bloc range le bloc de transformation avec la plus petite (***rms***) devient une partie de la transformation d'image sans se soucier comment le range peut être couvert. Dans un algorithme adaptatif l'erreur d'un seul bloc de transformation est comparée à une erreur maximale prédéfinie appelée erreur de collage Δ. Si l'erreur calculée est plus grande le bloc range spécifique est divisé en des blocs qui sont alors couverts indépendamment. Donc par rapport à l'algorithme non adaptatif, une qualité d'image peut être garantie. La compression fractale est non symétrique.

L'étape de codage a un coût de programmation énorme qui dépend du nombre des blocs domaines à comparer avec chaque bloc range, par exemple une image de taille n x n avec des domaines carrées de taille h x h nécessitent $(n-h+1)^2$ domaines. En outre chaque bloc domaine a 8 isométries. Donc chaque bloc range doit être comparé à $8(n-h+1)^2$ blocs domaines. Pour une image de taille 256 x 256, les blocs ranges ont la taille 8 x 8 et les blocs domaines ont la taille 16 x 16, l'algorithme effectue 516104192 opérations de calcules de ***rms***. Ce qui rend l'étape de codage plus lente par rapport à l'étape de décodage.

Cette énorme complexité de calcul montre clairement le besoin pour accélérer la vitesse d'exécution.

Une solution de ce problème est la classification des blocs. Cette méthode est un moyen efficace pour réduire le nombre de comparaisons lors de la recherche de transformations. A la création de « la domain pool », chaque domaine est rangé dans une classe. Lors de la recherche, le range est à son tour classé et les comparaisons ne se feront qu'avec les domaines se trouvant dans la même classe. La difficulté est bien entendue de trouver le critère de classification raisonnable, tel que la probabilité que le meilleur domaine ne se trouve pas dans la même

classe que le range soit minimisée. Eventuellement, si leur nombre est assez grand, plusieurs classes peuvent être examinées pour chaque range.

Il y a deux méthodes de classification très répandues :

- **Le schéma de classification de Fisher**

  Cette technique donne de meilleurs résultats lorsque 24 classes sont examinées, mais le facteur d'accélération est alors seulement à 3, ce qui limite considérablement l'intérêt de cette classification [6].

- **Le schéma de classification de Jacquin**

  Le problème de cette technique est que le nombre de classes est petit (3 clases) et que lorsqu'on tente de l'augmenter en créant des classes intermédiaires, la précision devient moins bonne [6].

La complexité liée à l'approche séquentielle et les problèmes de la classification justifient l'utilisation d'une autre approche qu'est la parallèlisation d'algorithmes qui sera présentée dans la section suivante.

6.4 L'approche parallèle de la compression fractale

D'après la littérature, une grande quantité de travaux ont été consacrés aux algorithmes pour les architectures MIMD [39].

Nous avons choisi une architecture MIMD à mémoire distribuée pour les raisons suivantes :

- elle est basée sur les techniques et les technologies modernes (actuelles).
- elle est la plus utilisée par rapport aux autres architectures.
- convient bien à la parllèlisation via les ranges.
- on ne peut pas effectuer la parallèlisation via les ranges sur une architecture SIMD (à mémoire distribuée) à cause des contraintes de la mémoire [39].

On identifie deux groupes majeurs d'algorithmes qui sont utilisés pour les calculs sur les architectures MIMD et qui sont appliqués en fonction de la relation entre la capacité de la mémoire d'un PE et la taille de l'image [39].

- **Algorithmes classe A :** Parallèlisation à travers les ranges

  Au sein de cette classe, il est au moins nécessaire que la totalité de l'image peut être stockée dans la mémoire d'un PE. En dehors des données de l'image

« la domain pool » peut être produite complètement. A chaque PE un sous-ensemble de « la range pool » est affecté ou bien statiquement ou dynamiquement et le PE calcule les meilleures transformations pour son sous-ensemble. Pour plus de détail voir [39] [40] et [41].

- **Algorithmes classe B :** Parallèlisation à travers les domaines

Au sein de cette classe d'algorithmes « la domain pool » ne peut être stockée dans la mémoire du PE, par conséquent « la domain pool » est distribuée uniformément sur les Pes. L'algorithme le plus efficace doit être opéré à la manière « pipeline » dans l'ordre d 'effectuer les comparaisons range-domaine de tous les sous-domain pool [39]. Pour plus de détail voir [40].

6.5 Parallèlisation de l'algorithme de compression fractale

6.5.1 Les algorithmes séquentiels

Il existe plusieurs algorithmes séquentiels, chacun utilise une technique différente. Voici quelques algorithmes :

- **Algorithme 1 :** (Algorithme de base, non adaptatif)

Partitionner l'image originale en ranges R_i.
Pour chaque R_i Faire
  Pour chaque D_j de la domain pool Faire
    Calculer la transformation $D_j \rightarrow R_i$
    Si la transformation est meilleure Alors
      Associer R_i à D_j et à la transformation correspondante
    FinSi
  Fin pour
  Sauvegarder la transformation w_i.
Fin pour

Cet algorithme est non adaptatif, c'est l'algorithme de base. La partition est fixe et les ranges ont la même taille (en général de taille 8 x 8), les domaines aussi ont la même taille (en général 16 x 16).

Pour chaque range on cherche la meilleure erreur *rms*, puis on stocke la transformation correspondante. A la fin nous obtenons le code IFS.

Le résultat est étonnamment bon, au regard de la simplicité de l'algorithme de codage[5].

- **Algorithme 2 :** (adaptatif)

Choisir un niveau de tolérance e_c.

Mettre $R_1 = I^2$ et le marquer comme non couvert.

Tant qu'il y a des ranges non couverts R_i Faire

Parmi les domaines possibles D, trouver le domaine D_i et le w_i

Correspondant qui couvrent le mieux R_i

Si l'erreur rms < e_c ou taille(R_i) <= r_{min} Alors

Marquer R_i comme couvert, et sauvegarder la transformation w_i

Sinon

Partitionner R_i en des ranges plus petits, qui sont marqués non couverts,

Et enlever R_i de la liste des ranges non couverts.

FinSi

FinT.Q

Cet algorithme est adaptatif, il cherche des transformations qui répondent à une fidélité e_c.

- **Algorithme 3 :** (adaptatif)

Choisir un nombre N_r de ranges à trouver.

Initialiser une liste à $R_1 = I^2$, et marquer ce range comme non couvert

Tant qu'il y a des ranges non couverts dans la liste Faire

Pour chaque range non couvert de la liste, trouver et stocker le domaine

D_i Є D et la transformation w_i qui le couvre le mieux, et marquer ce range

comme couvert

Dans la liste des ranges, trouver le range R_j avec taille(R_j) > r_{min}

qui a la plus grande erreur *rms* (qui est le moins bien couvert)

Si le nombre de ranges dans la liste est inférieur à N_r Alors

Partitionner R_j en des ranges plus petits, qui sont ajoutés à la

liste et marqués non couverts.

Enlever R_j, w_j et D_j de la liste.

FinSi

FinT.Q

Sauvegarder tous les w_i de la liste.

Cet algorithme est adaptatif, il fait la compression avec N transformation.

- **Algorithme 4 :** (Triangulation)

```
Calculer la triangulation R
Calculer la triangulation D
Pour  I allant de 1 à N (I=indice de triangle r_i de R)
  {
    erreur_min = float_maximum
    pour  j allant de 1 à Q (j = indice du triangle d_j de D)
      {
        erreur = d(r_i, w(d_j))
        si erreur < erreur_min alors mémoriser
          {
            la combinaison du collage de d_j sur r_i
            le coefficient d'échelle beta_2(j) optimal
            le coefficient de décalage beta_1(j) optimal
            j_min = j
            erreur_min = erreur
          }
      }
    stocker {
                l'indice j_min
                la combinaison du collage de d_j_min sur r_i
                le coefficient d'échelle beta_2(j_min) optimal
                le coefficient de décalage beta_1(j_min) optimal
            }
  }
```

Cet algorithme est basé sur le partitionnement en triangles.

6.5.2 Parallèlisation de l'algorithme

Nous avons choisi l'**algorithme 1** qui est l'algorithme de base non adaptatif.
Nous avons choisi cet algorithme puisque sa programmation et sa parallèlisation sont simples au contraire des algorithmes adaptatifs (quadtree) et de triangulation

qui sont un peut complexes. Ainsi que notre principal objectif est de montrer que la parallèlisation réduit le temps de codage.

L'approche de parallèlisation que nous avons adopté est la classe A d'algorithmes parallèles à cause de sa simplicité de programmation ainsi qu'elle nécessite moins de communications entre les PEs.

Cet algorithme sera parallélisé sur une architecture parallèle MIMD à mémoire distribuée comme suit :

Algorithme :

Assigner un sous-ensemble de la range-pool à chaque PE.
Sur chaque PE Faire
  Pour chaque range R_i Faire
    Pour chaque domaine 'D_j' Faire
      Calculer la meilleure transformation 'D_j' → 'R_i'
      Si la transformation est meilleure Alors
        Associer 'R_i' à 'D_j' et à la transformation correspondante
      FinSi
    Fin pour
  Fin pour
Fin

Cet algorithme est conçu pour une machine parallèle à la base d'une architecture massivement parallèle MIMD à mémoire distribuée.

Pour chaque processeur élémentaire un sous-ensemble de ranges lui est associé.
En parallèle, chaque processeur cherche la meilleure transformation pour tous les ranges du sous-ensemble qui lui est associé en se basant sur la meilleure *rms* calculée, et associe la transformation correspondante à chaque range.

Pour des raisons de disponibilité et de coût, nous avons retenu comme machine MIMD à mémoire distribuée un réseau Ethernet. Pour cette raison, l'algorithme proposé est adapté pour fonctionner sur le réseau Ethernet en le structurant en deux algorithmes, le 1er en tant que serveur (Host) et le second en tant que client (Node).

Serveur (Host) : pour un réseau Ethernet

Etablir la connexion avec les clients.
Charger l'image.
Envoyer les données initiales (image, taille de l'image) aux clients.
Termine = faux
Tant que non(Termine) Faire
  Bloquer jusqu'à un client envoi un signal de fin de codage.
  Marquer le client i comme terminé.
  Nbt = Nbt + 1 ;
  Si Nbt = Nombre de clients Alors
    Termine = vrai
  FinSi
FinT.Q
Pour chaque client Faire
  Collecter les résultats à partir du client
  et terminer le client
Fin pour
Ecrire le résultat complet du codage fractal dans un fichier.
FIN

Après l'établissement de la connexion entre le serveur et les clients, le serveur charge l'image à compresser et l'envoi aux différents clients avec sa taille.
Dans l'étape suivante, le serveur se met en attente (écoute) pour recevoir les signaux de fin de codage émis par les différents clients. Si un client envoi un signal de fin de codage, le serveur le considère comme client terminé et incrémente un compteur qui indique le nombre de clients ayant terminé le codage.

Une fois tous les clients ont terminé leur travail, le serveur collecte les résultats du codage à partir des différents clients séquentiellement pour conserver l'ordre des transformations correspondantes aux ranges dans l'image. Après la fin de la réception de toutes les transformations dans l'ordre correspondant, il écrit le résultat du codage de l'image dans un fichier avec un code fractal.

Client (Node) : pour un réseau Ethernet

Recevoir les données initiales (image, taille de l'image)
Calculer le sous-ensemble de ranges correspondant au client.
Pour (chaque range) Faire
  Pour (chaque domaine) Faire
    Pour (chaque isométrie du domaine D_j) Faire
      Calculer_rms(range R_i, domaine D_j)
      Si rms < meilleure_rms jusqu'à maintenant Alors
        Meilleure_rms = rms
      FinSi
    Fin pour
  Fin pour
  Sauvegarder la transformation correspondante au meilleur *rms* range-domaine
Fin pour
Envoyer un signal de fin de codage au serveur.
Bloquer jusqu'à ce que le serveur envoi un signal pour l'envoi du résultat.
Envoyer les transformations obtenues pour le sous-ensemble de ranges.
FIN

Après que le client i reçoit l'image et sa taille, il calcule le sous-ensemble de ranges qui lui correspond, ensuite il commence l'opération de codage. Pour chaque range du sous-ensemble, le client cherche la meilleure transformation correspondante à la meilleure erreur *rms* et la sauvegarde.
Lorsque tous les ranges sont traités et leurs transformations sont calculées, le client envoi un signal de fin de codage au serveur et il se met en état d'écoute jusqu'à la réception d'un signal émanant du serveur l'ordonnant de lui envoyer le résultat (l'ensemble de transformations) au serveur. Si c'est le cas le client envoi les transformations calculées au serveur.
La décomposition de l'algorithme parallèle en deux algorithmes serveur et client pour qu'il soit adapté à un réseau permet de réduire le temps de codage d'une manière intéressante par rapport à celui de l'algorithme séquentiel et ce selon le nombre de postes utilisés, c. à. d que le temps de calcul est divisé par le nombre

de postes. C'est ça que nous allons montrer par une implémentation de cet algorithme.

6.6 Conclusion

Dans ce chapitre, nous avons montré que la compression d'images fixes par la méthode fractale nécessite une énorme complexité de calcul. Pour accélérer la vitesse d'exécution, plusieurs approches ont été proposées : la classification et la parallèlisation.

La classification est une méthode efficace mais elle a ses limites.

La parallèlisation est la méthode que nous avons adopté pour réduire le temps de compression. Nous avons choisi l'architecture MIMD à mémoire distribuée comme support pour l'application de notre algorithme qui représente l'algorithme de base. Nous avons parallélisé l'algorithme séquentiel selon l'approche suivante : parallèlisation à travers les ranges.

Pour des raisons de disponibilité et de coût, nous avons proposé deux algorithmes : serveur et client qui sont adaptés à un réseau Ethernet qui est bien adapté à une architecture MIMD à mémoire distribuée.

Le chapitre suivant est consacré à l'implémentation de notre algorithme et aux résultats obtenus.

Chapitre 7

Implémentation et résultats

Le chapitre précédent a été consacré à la parallèlisation de l'algorithme séquentiel de base de la compression par fractales. Dans ce chapitre nous allons décrire l'implémentation de l'algorithme parallèle (l'application) et de présenter les résultats obtenus.

7.1 Implémentation parallèle du codage fractal

La parallèlisation à travers les ranges est l'approche que nous avons adopté pour réaliser notre application. Chaque sous-ensemble de la range-pool est associé à un pc client avec toute la domain-pool. Les principales raisons de ce choix :

- Chaque range subit un traitement (comparaison avec les domaines possibles) pour trouver la meilleure transformation d'une manière totalement indépendante des autres ranges, ce qui élimine totalement la synchronisation et la communication entre les processus qui codent des ranges différents sur les autres pc clients.
- Cette technique est déjà utilisée dans plusieurs travaux antérieurs [42].

7.1.1 Langage de programmation

Borland C++Builder est un environnement de programmation visuelle orienté objet permettant le développement rapide d'applications Windows 32 bits. En utilisant C++Builder, vous pouvez créer de puissantes applications Windows avec un minimum de programmation.

C++Builder propose une bibliothèque de classes complète appelée VCL (bibliothèque de composants visuels) et une suite d'outils de conception rapide d'applications (RAD), y compris des modèles d'application et de fiche, ainsi que des experts de programmation. C++Builder gère réellement la programmation orientée objet : la bibliothèque de classes comporte des objets qui encapsulent l'API Windows ainsi que des techniques de programmation utiles.

Nous avons choisi C++Builder comme un langage de programmation pour le développement de notre environnement parallèle pour la compression d'images fixes à l'aide des fractales puisqu'il offre les moyens, les composants et les méthodes nécessaires et efficaces pour l'envoi et la réception des données et l'établissement des connexions entre le serveur et les clients et aussi par ce qu'il nous permet de développer une interface graphique puissante et simple à utiliser.

C++Builder offre les composants sockets qui permettent d'établir des connections et d'échanger les données entre plusieurs postes de travail dans un réseau.

Ces composants sont les suivants : Un composant socket client *(TClientSocke*t) qui transforme l'application en client TCP/IP. Les sockets client vous permettent de spécifier le socket serveur auquel vous souhaitez vous connecter et le service que vous attendez de ce serveur. Lorsque vous avez décrit la connexion voulue, vous pouvez utiliser le composant socket client pour établir la connexion au serveur. Le deuxième est un composant socket serveur *(TServerSocke*t) qui transforme l'application en serveur TCP/IP. Les sockets serveur vous permettent de spécifier le service que vous offrez ou le port que vous utilisez pour écouter les requêtes client. Vous pouvez utiliser le composant socket serveur pour écouter et accepter les requêtes de connexion client.

Chaque composant offre des méthodes de communication pour communiquer avec l'autre extrémité de la connexion.

Pour lire sur la connexion socket, utilisez les méthodes *ReceiveBuf* ou *ReceiveTex*t. Avant d'utiliser *ReceiveBu*f, utilisez la méthode *ReceiveLength* pour avoir une estimation du nombre d'octets que le socket à l'autre bout de la connexion est prêt à émettre. Pour écrire sur la connexion socket, utilisez la méthode *SendBu*f ou *SendTex*t.

En plus de ça, C++Builder facilite la programmation de ce type d'application grâce aux différents types d'événements de notification qu'il offre. Ces événements permettent à l'application d'effectuer différentes tâches selon l'événement (envoi de données, réception de données, déclenchement d'une opération, etc…).

Les sockets génèrent des événements de lecture et d'écriture qui informent votre socket de la nécessité de lire ou d'écrire via la connexion. Avec les sockets client, vous pouvez répondre à ces notifications dans un gestionnaire d'événement

OnRead ou *OnWrite*. Avec les sockets serveur, vous pouvez répondre à ces notifications dans un gestionnaire d'événement *OnClientRead* ou *OnClientWrite*.

7.1.2 Architecture matérielle

Pour des raisons de disponibilité et de coût, nous avons retenu comme machine MIMD à mémoire distribuée un réseau Ethernet. Le réseau est constitué d'un ensemble de nœuds, chaque nœud c'est un PC de type AMD Durron (1.3 GHz) et il dispose d'une mémoire vive de taille 128 Mo. Pour comprendre le fonctionnement d'un réseau Ethernet, nous l'introduirons dans le paragraphe qui suit.

7.1.2.1 Ethernet

Ethernet est une technologie universelle qui dominait déjà les réseaux locaux bien avant le développement de l'Internet. Ethernet était à l'origine un standard développé par les laboratoires Xerox au tout début des années 1970. Ce standard a d'abord évolué jusqu'à la version Ethernet II aussi appelée DIX ou encore v2.0 avec l'association regroupant Digital Equipment Corporation, Intel et Xerox. Par la suite, Ethernet a été inclus dans les travaux sur la modélisation OSI au début des années 1980. Depuis cette époque, la technologie Ethernet est totalement indépendante des constructeurs ; c'est un des facteurs importants de sa popularité[43].

7.1.2.2 Principes de base

- Toutes les stations sont égales vis-à-vis du réseau : il n'y a pas d'équipement maître de contrôle du réseau.
- La méthode d'accès employée est distribuée entre tous les équipements connectés.
- Le mode de transmission est de type bidirectionnel à l'alternat : les signaux transitent dans les deux sens, mais pas simultanément.
- On peut relier ou retirer une machine du réseau sans perturber le fonctionnement de l'ensemble.

7.1.2.3 Evolution de réseau Ethernet

La simplicité de la méthode d'accès et la simplicité de l'interconnexion avec les autres technologies ont fait d'Ethernet une technologie évolutive à des coûts acceptables pour toutes les catégories d'utilisateurs.

Même si les câbles coaxiaux ont été abandonnés lors du passage de 10 à 100Mbps, c'est cette simplicité de mise en œuvre qui permet une évolution progressive vers les technologies multimédias à partir des infrastructures existantes sans réinvestissements lourds.

Les câblages en paires torsadées de catégorie 5 (UTP 5) sont utilisables de 10 Mbps jusqu'à 1 Gbps.

Aujourd'hui l'appellation Ethernet regroupe trois familles:

- Ethernet et IEEE802.3 : la définition d'origine à 10Mbps[44],
- FastEthernet l'extension à 100Mbps,
- GigabitEthernet l'extension à 1000Mbps ou 1Gbps[43].

A l'intérieur de chaque famille, il existe de nombreuses déclinaisons.

Dans notre cas, nous avons utilisé un réseau de type FastEthernet. Les nœuds sont liés par des câbles de type RJ45 en utilisant un Switch. Ici nous avons adopté la topologie étoile puisqu'elle permet une connexion de type point-à-point entre les différents nœuds de réseau. La structure de réseau est montrée dans la figure 7.1.

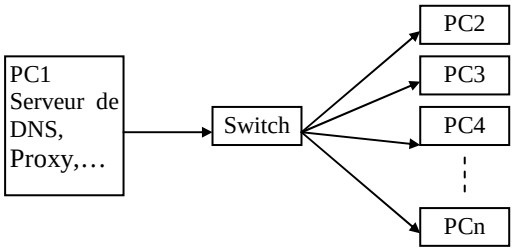

Figure 7.1 : Réseau Ethernet (topologie étoile)

7.1.2.4. Configuration du réseau

La communication dans notre modèle est basée sur l'utilisation des routines de la norme TCP/IP. TCP/IP est issue de recherches effectuées par DARPA (Defense Advanced Projects Agency) pour réaliser un réseau fournissant des connexions sur une bande passante large entre les principaux sites informatiques gouvernementaux (USA). TCP/IP est étroitement lié à Ethernet, à tel point que les deux mots étaient devenus pratiquement interchangeables.

Dans notre réseau, chaque machine doit avoir une adresse différente. De manière à ce que l'information qui lui est destinée lui soit effectivement livrée. C'est l'Internet Protocol (IP) qui contrôle ce schéma d'adressage.

7.1.3 Outils logiciels

Comme nous l'avons évoqué dans le paragraphe précédent, les méthodes de communication utilisées dans notre logiciel sont basées sur l'utilisation des protocoles du système TCP/IP. TCP/IP est indépendant de toute plate forme logicielle et matérielle. C'est à dire, nous pouvons utiliser n'importe quel système d'exploitation (Unix, Linux, Windows,…). Dans notre implémentation, nous avons installé sur chaque poste un système Windows 98. Nous avons aussi exploité les protocoles du système TCP/IP à travers le pacquage Internet de l'environnement C++Builder.

7.1.4 Modèle d'exécution

La parallèlisation des algorithmes de compression par blocs d'image sur architecture MIMD à mémoire distribuée convient bien sur l'organisation Maître/Esclaves [45]. Le maître tient l'état général de l'application et possède une vue globale sur la progression de la compression. C'est lui qui annonce le début et la fin du codage. Son rôle consiste à effectuer les tâches séquentielles de l'algorithme de compression et distribuer les blocs de l'image sur les esclaves. Ces derniers codent les blocs (les ranges) d'image qu'ils ont reçu et renvoient les résultats au maître. De cette façon ils possèdent seulement une vue partielle sur la progression de la compression puisque chaque esclave ne voit que les tâches qui lui ont été assigné (figure 7.2). Après la réception de tous les blocs (les ranges) compressés, le maître commence à construire le Codestream, le fichier final de l'image compressée. Cette tâche est séquentielle et nécessite la disponibilité de tout les ranges codés (toutes les transformations).

7.1.5 Outil de communication

Nous avons utilisé les Sockets, étant donné que notre principal objectif est la minimisation du temps d'exécution. Les Sockets représentent un mécanisme de communication de bas niveaux qui permet d'accéder directement à l'interface réseau. Les sockets offrent un mécanisme de hautes performances.

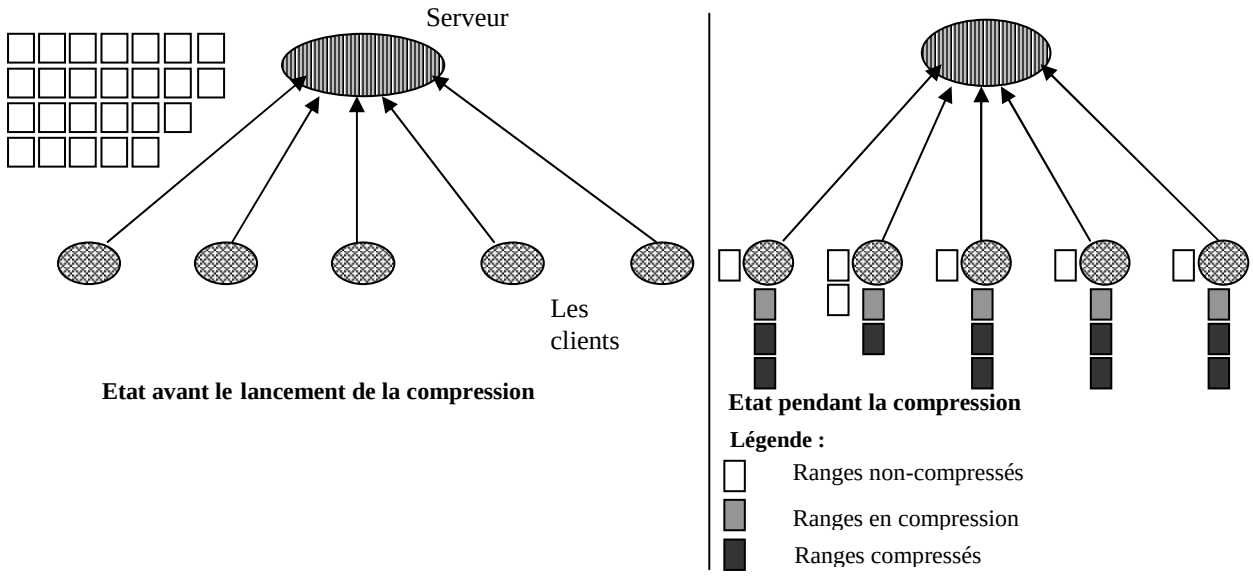

Figure 7.2 : Modèle d'Exécution Maître/Esclave.

7.1.5.1 Déroulement d'une communication

Les nœuds communiquent en s'envoyant des messages. Nous allons examiner le déroulement d'une communication point-à-point, c'est à dire entre deux nœuds. Cette communication se déroule en deux temps :

- Envoi d'un message : le processus détenant des données à communiquer les envois explicitement, en précisant le destinataire. Les données sont placées dans un tampon d'émission, et sont ensuite acheminées sur le réseau ;
- Réception du message : le processus destinataire reçoit le message. Les données reçues du réseau sont placées dans un tampon.

7.1.5.2 Les méthodes d'envoi

Les méthodes d'envoi que nous avons utilisé sont les suivantes :

1. La méthode SendBuf : elle permet d'écrire dans la connexion de socket. Elle est utilisée pour envoyer à un hôte distant un tampon de données.

   int __fastcall SendBuf(void *Buf, int Count);

   Count : est le nombre d'octets à écrire dans la connexion de socket.

   Buf : est le buffer qui contient les données à écrire dans la connexion de socket.

   ServerSocket->Socket->Connections[I]->SendBuf(m_image, 8192);

Cette commande permet au serveur l'envoi au client I (I : le numéro du client) un bloc d'image contenu dans le buffer m_image qui est un tableau à deux dimensions, ce buffer est de taille 8192 octets.

ClientSocket->Socket->SendBuf(Resulta, 2048);

Cette commande permet au client l'envoi au serveur un bloc résultat du codage contenu dans le buffer Resulta qui est un tableau à une dimension, ce buffer est de taille 2048 octets.

2. La méthode SendText : elle permet d'écrire dans la connexion de socket. Elle est utilisée pour envoyer à un hôte distant une chaîne.

   int __fastcall SendText(const AnsiString S);

   S: contient la chaîne à écrire dans la connexion de socket. Ce paramètre est de type AnsiString.

   ServerSocket->Socket->Connections[I]->SendText("S");

   Cette commande permet au serveur l'envoi au client I le caractère S pour lui demander de commencer l'envoi des résultats.

   ClientSocket->Socket->SendText("T");

   Cette commande permet au client l'envoi au serveur le caractère T pour lui informer qu'il a terminé le codage.

7.1.5.3 Les méthodes de réception

Les méthodes de réception que nous avons utilisé sont les suivantes :

1. La méthode ReceiveBuf : elle est utilisée pour lire des données via la connexion de socket et les stocke dans un tampon. Avant d'appeler cette méthode, il est demandé d'appeler la méthode ReceiveLength qui renvoi la taille estimée du tampon nécessaire pour récupérer des informations du socket.

   int __ fastcall ReceiveBuf(void *Buf, int Count);

   Count : est le nombre d'octets à lire de la connexion de socket.

   Buf : est le buffer qui contient les données lues de la connexion de socket.

   ServerSocket->Socket->Connections[I]-> ReceiveBuf(Resulta, 2048);

   Cette commande permet au serveur de recevoir des données(résultats) envoyées par le client I et les stocke dans le buffer Resulta qui est un tableau à une dimension, ce buffer est de taille 2048 octets.

   ClientSocket->Socket->ReceiveBuf(m_image, 8192);

Cette commande permet au client de recevoir un bloc d'image envoyé par le serveur et le stocke dans le buffer m_image qui est un tableau à deux dimensions, ce buffer est de taille 8192 octets.

2. La méthode ReceiveText: elle est utilisée pour lire une chaîne via la connexion de socket.

   AnsiString __ fastcall ReceiveText();

   Socket->ReceiveText();

   Cette commande permet de lire une chaîne de la connexion envoyée par un socket serveur ou client.

7.1.6 Structures de données

Les structures de données sont des pointeurs sous forme de tableaux d'une seule dimension ou de deux dimensions. Ces tableaux contiennent les données de l'image (toute l'image ou une partie de l'image) ou des résultats intermédiaires des traitements faits sur les ranges de l'image. Il y a aussi des structures de données de type **struct** (enregistrement) qui contiennent les paramètres de la transformation.

7.1.7 Fonctionnement de l'application

Puisqu'il n'existe pas d'équipement maître de contrôle du réseau, nous avons développé notre modèle de telle sorte qu'une seule station supervise toutes les autres stations. Mais chaque machine a le droit d'être la station maîtresse. Nous avons développé deux applications, une de type serveur et l'autre de type client. Une copie de l'application cliente (Esclave) est installée sur chaque machine de réseau. L'application serveur (Maître) est installée sur une seule machine.

7.1.7.1 Fonctionnement du serveur

Le serveur reste en écoute des requêtes des clients qui veulent se connecter à ce dernier. Lorsque tous les clients ont été connectés au serveur, ce dernier commence l'envoi des données initiales. Au premier temps, il envoi la taille de l'image et le nombre des postes (clients) qui est une information utile aux clients. Ensuite, il décompose l'image en blocs et les envoi l'un après l'autre aux clients.

Le serveur envoi le premier bloc au premier client et reste en état d'attente jusqu'à ce que le client envoi un signal ' R' indiquant au serveur qu'il a reçu le

bloc d'image envoyé par ce dernier. Le serveur envoi le deuxième bloc et le même processus se répète jusqu'à l'envoi de tous les blocs constituant l'image.

Le serveur répète cette opération pour tous les clients d'une manière séquentielle.

Les clients commencent le codage et le serveur reste en état d'attente jusqu'à ce que les clients terminent le codage. Le serveur reçoit un signal de fin de codage 'T' envoyé par un client qui a terminé son travail, le marque comme client terminé et incrémente un compteur des clients ayant terminé le codage.

Une fois tous les clients ont terminé le codage, le serveur envoi d'une manière séquentielle (en respectant le même ordre d'envoi de l'image) à chaque client un signal 'S' qui lui demande d'envoyer le résultat. Il reçoit tous les résultats des clients séquentiellement.

Lorsque la collection des résultats à partir des clients est terminée, il commence la tâche qu'est la construction du fichier contenant le code fractal (IFS) de l'image compressée (voir diagramme 7.1).

Le temps total de la compression est mesuré de la manière suivante : Avant le début d'envoi des données initiales, on fait appel à une instruction « gettime() » qui permet de prélever l'heure du début qui a la forme suivante : hh : mm : ss : msms.

A la fin de la collection des résultats et la construction du fichier contenant le code IFS, on fait un autre prélèvement de l'heure de la fin du codage qui a la même forme que celle du début puis on fait la différence entre l'heure de la fin et l'heure du début pour obtenir le temps total du codage sur le serveur.

7.1.7.2 Fonctionnement du client

Après qu'il reçoit la taille de l'image et le nombre de postes clients qu'est une information utile pour accomplir son travail, le client se met en état d'attente pour recevoir les données de l'image sous forme de blocs.

Lorsque le client reçoit un bloc, il le stocke dans la place correspondante dans un tableau à deux dimensions, puis il envoi un signal 'R' au serveur lui indiquant qu'il a reçu le bloc envoyé et lui demande l'envoi d'un autre bloc. Ce processus se répète jusqu'à ce que tous les blocs constituants l'image ont été réceptionnés par le client. Ensuite, le processus de codage se déclenche.

A la fin du codage, le client envoi un signal de fin de codage ‘T’ au serveur et il attend un signal ‘S’ qui lui demande d’envoyer le résultat du codage au serveur. Le travail du client se termine après l’envoi du résultat au serveur (voir diagramme 7.2).

Le temps de codage concernant le client est mesuré de la manière suivante : Avant le début du codage, on fait appel à l’instruction « gettime() » qui permet de prélever l’heure du début de codage qui a la forme suivante : hh : mm : ss : msms.

A la fin du codage, on fait un autre prélèvement de l’heure de fin du codage qui a la même forme que celle du début puis on fait la différence entre l’heure de la fin et l’heure du début pour obtenir le temps du codage du client.

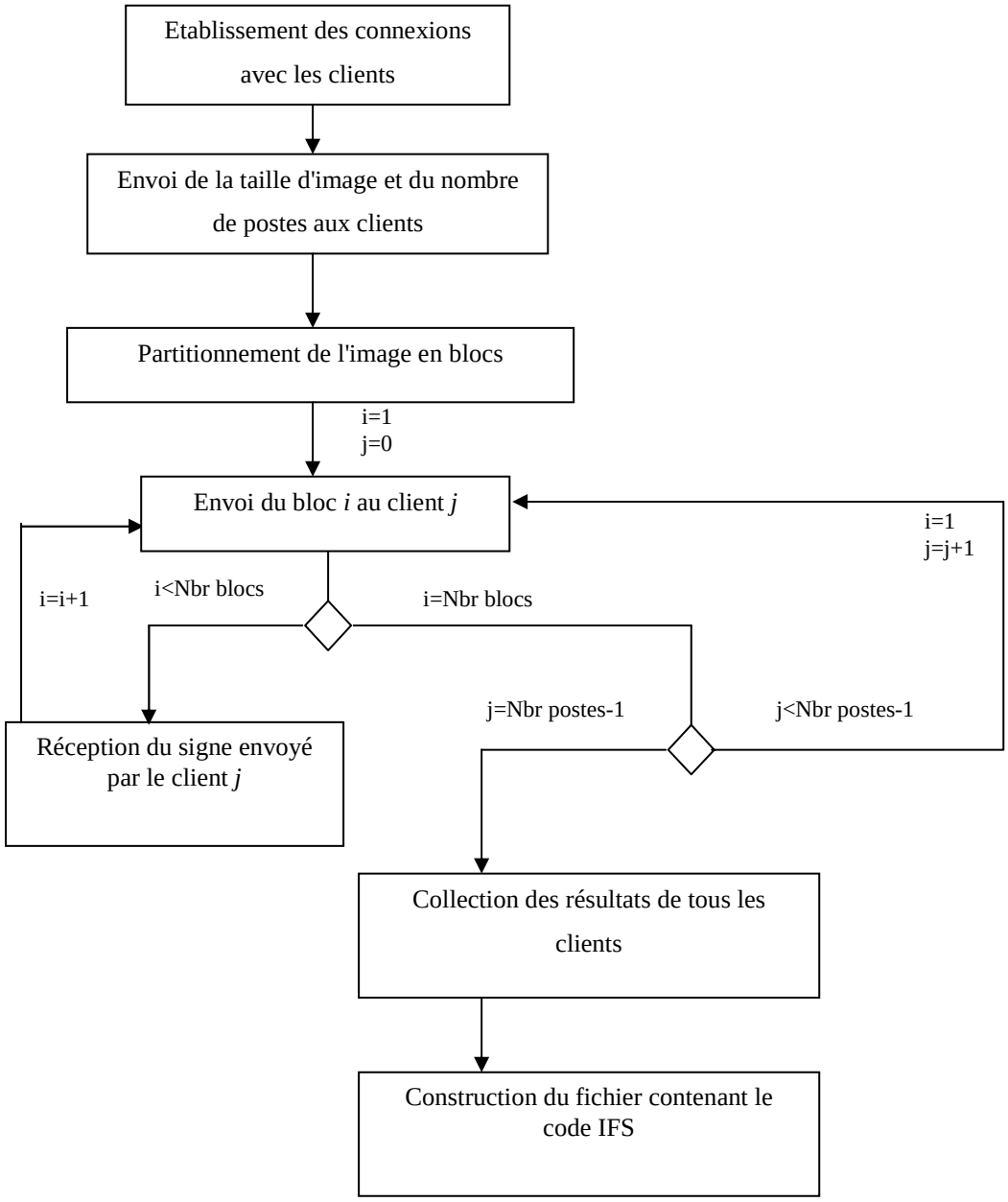

Diagramme7.1: fonctionnement du serveur

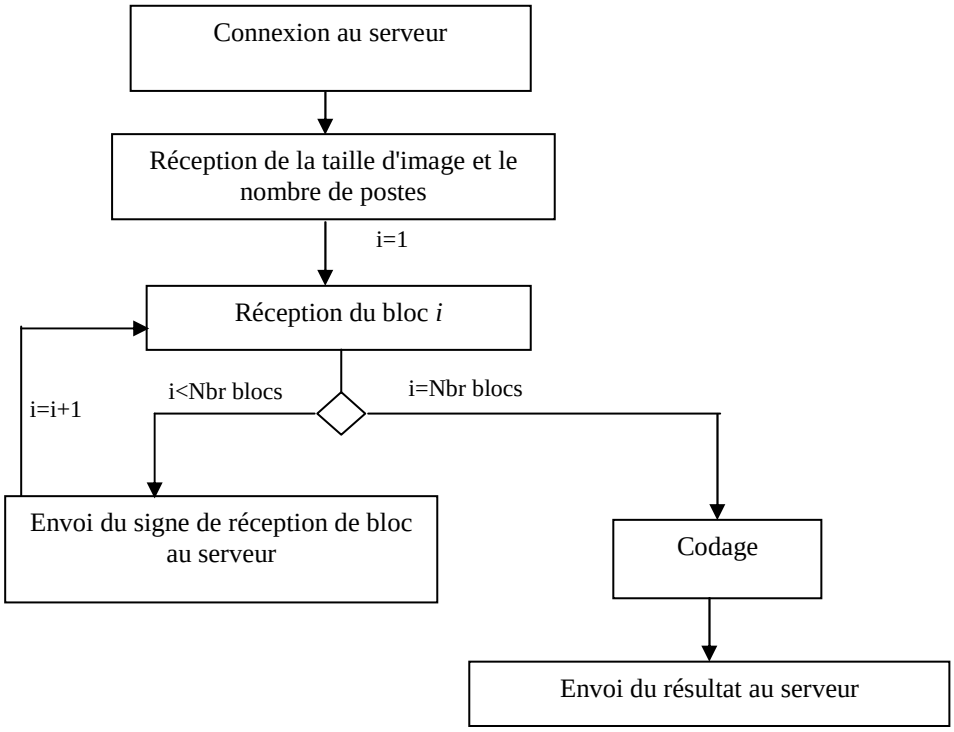

Diagramme7.2: fonctionnement du client

7.1.7.3 Description des différentes méthodes de l'application

1. Le Serveur

L'application serveur contient les méthodes et procédures principales suivantes :

Timer1Timer () : elle est responsable des tâches suivantes :

- envoi de la taille de l'image.
- envoi du nombre de postes.
- envoi du premier bloc de l'image au bloc.

AllocCharg() : elle est responsable des tâches suivantes :

- allocation de la mémoire dynamique qui contient l'image à charger du fichier et les résultats.

SpeedButton5Click () : elle est responsable des tâches suivantes :

- envoi le numéro de chaque client correspondant.
- déclenche le gestionnaire associé à l'objet Timer1.

ServerSocketCompClientRead() : elle est responsable des tâches suivantes :

- envoi des blocs restant de l'image aux différents clients.
- reçoit et collecte les résultats à partir des différents clients.

- envoi et réception des signaux qui permettent la communication entre le serveur et les clients pour l'envoi et la réception des données et des résultats.

EnvoiBloc() : elle est responsable des tâches suivantes :

- permet l'envoi d'un bloc, elle est appelée par la méthode : ServerSocketCompClientRead()

En plus de ces méthodes, il y a d'autres méthodes qui assurent le processus de connexion des clients au serveur.

2. Le Client

L'application Cliente contient les méthodes et procédures principales suivantes :

ClientSocketCompRead() : elle est responsable des tâches suivantes :

- Reçoit le numéro du client, la taille de l'image et le nombre de postes.
- Reçoit les blocs de l'image envoyés par le serveur.
- Envoi le résultat du codage lorsque le serveur le demande.
- Envoi et réception des signaux qui permettent la communication entre le serveur et les clients pour l'envoi et la réception des données et des résultats.

AllocCharg() : elle est responsable des tâches suivantes :

- Allocation de la mémoire dynamique qui contient l'image envoyée par le serveur et le résultat du codage fait par le client.

Timer1Timer () : elle est responsable des tâches suivantes :

- Déclenche le processus de codage.
- Compléter l'envoi du résultat divisé en blocs lorsque la taille du bloc est plus grande.

Compress() : elle est responsable des tâches suivantes :

- Codage des ranges et production des transformations correspondantes
- Elle fait appel à la fonction Delta().

delta() : elle est responsable des tâches suivantes :

- calcul de l'erreur rms et les paramètres de la transformation (s, o, isométrie).

rotation() : elle est responsable des tâches suivantes :

- renvoi une isométrie d'un domaine parmi les huit possibles.

En plus de ces méthodes, il y a d'autres méthodes qui assurent le processus de connexion du client au serveur.

7.2 Expérimentation et discussion

7.2.1 Conditions d'expérimentation

Notre application a été implémentée et testée sur un réseau de stations de travail dont la configuration est la suivante :

- Les nœuds du réseau sont des machines AMD Durron(1.3 GHz, 128 Mo de RAM).
- Système d'exploitation : Windows 98.
- C++Builder version 5 de Borland.
- Réseaux : Ethernet (10BaseT).

Nous avons testé notre application sur les images : Lena, Collie et San Fransisco (voir annexe C) de différentes tailles (256 x 256, 256 x 512, 512 x 256, 512 x 512) en jouant sur le nombre d'esclaves (nombre de clients).

7.2.2 Expérimentation

Les tableaux suivants (7.1, 7.2, 7.3, 7.4) contiennent les résultats détaillés des mesures du temps de compression que nous avons faits. Les tests ont été effectués pour des images de différentes tailles en jouant sur le nombre de postes (clients) (1, 2, 4, 8).

Nous remarquons que le temps de compression diffère d'un client à un autre et que le temps total de compression sur le serveur est lié au temps du dernier client qui termine le codage.

Théoriquement le temps de compression sur les clients est le même mais on remarque que cette différence est relative au système informatique utilisé pour effectuer les tests. Ces derniers ont été effectués sur des machines AMD Durron (1.3 GHz et 128 Mo de RAM) doté du système windows 98 qui ont été réservé uniquement pour effectuer les tests afin d'éliminer tous les facteurs qui peuvent influencer le résultat.

Un système informatique est composé de deux parties : le matériel (hardware) et le système d'exploitation (software). Ces deux facteurs sont la cause de cette différence entre les clients puisque ces derniers effectuent les mêmes traitements sur le même nombre de ranges. Chaque client est lui est attribué un bloc de ranges de même taille que les autres clients.

Nous remarquons que cette différence de temps a influé sur les résultats obtenus.

Taille	LENA	COLLIE	SAN FRANSISCO
256 x 256	216 s 76 ms	210 s 91 ms	217 s 67 ms
256 x 512	876 s 44 ms	864 s 75 ms	876 s 1 ms
512 x 256	876 s 69 ms	864 s 47 ms	877 s 76 ms
512 x 512	3599 s 72 ms	3563 s 57 ms	3610 s 69 ms

Tableau 7.1 : Temps de compression séquentielle pour les images LENA, COLLIE et SAN FRANSISCO de différentes tailles (64, 128 et 256 Ko).

Le tableau 7.1 est constitué des rubriques suivantes:

La 1ère colonne concerne la taille des images sur lesquelles nous avons effectué les tests.

La 2ème colonne contient les temps de compression séquentielle des images de Lena.

La 3ème colonne contient les temps de compression séquentielle des images de Collie.

La 4ème colonne contient les temps de compression séquentielle des images de San fransisco.

Le tableau 7.2 contient les résultats en détaille de la compression parallèle des images de Lena.

Le tableau 7.3 contient les résultats en détaille de la compression parallèle des images de Collie.

Le tableau 7.4 contient les résultats en détaille de la compression parallèle des images de San fransisco.

Ces trois tableaux ont la même forme qui est la suivante:

La 1ère colonne concerne le nombre de postes clients.

La 2ème colonne concerne les numéros des postes clients pour un test selon leur nombre.

La 3ème colonne contient la taille de l'image sur laquelle a été effectué le test.

La 4ème colonne contient le temps global de compression parallèle (sur le serveur) pour un test bien déterminé (2, 4 ou 8 postes).

La 5ème colonne contient le temps de compression sur chaque poste client pour un test bien déterminé.

Les colonnes 6, 7 et 8 sont identiques aux colonnes 3, 4 et 5 et concernent les tests effectués sur d'autres images de taille différente.

Image de LENA							
Nb client	N° client	Taille	Temps par Serveur	Temps par Client	Taille	Temps par Serveur	Temps par Client
2	0	256 x 256	**125 s 47 ms**	88 s 27 ms	512 x 256	**469 s 28 ms**	361 s 91 ms
	1			116 s 87 ms			459 s 94 ms
4	0		**60 s 59 ms**	52 s 95 ms		**223 s 55 ms**	216 s 80 ms
	1			52 s 68 ms			216 s 8 ms
	2			44 s 44 ms			191 s 86 ms
	3			53 s 1 ms			214 s 82 ms
8	0		**33 s 50 ms**	26 s 64 ms		**120 s 18 ms**	109 s 56 ms
	1			22 s 8 ms			109 s 3 ms
	2			23 s 65 ms			90 s 19 ms
	3			26 s 42 ms			107 s 16 ms
	4			28 s 29 ms			114 s 69 ms
	5			23 s 40 ms			95 s 35 ms
	6			26 s 43 ms			103 s 4 ms
	7			27 s 24 ms			111 s 45 ms
2	0	256 x 512	**466 s 92 ms**	361 s 79 ms	512 x 512	**1834 s 55 ms**	1559 s 94 ms
	1			457 s 69 ms			1826 s 38 ms
4	0		**223 s 93 ms**	215 s 31 ms		**893 s 72 ms**	885 s 90 ms
	1			215 s 86 ms			884 s 74 ms
	2			191 s 86 ms			748 s 31 ms
	3			215 s 19 ms			880 s 34 ms
8	0		**120 s 7 ms**	104 s 5 ms		**460 s 22 ms**	395 s 17 ms
	1			108 s 96 ms			374 s 5 ms
	2			90 s 58 ms			371 s 74 ms
	3			108 s 31 ms			447 s 53 ms
	4			115 s 12 ms			449 s 62 ms
	5			91 s 1 ms			453 s 85 ms
	6			103 s 9 ms			393 s 5 ms
	7			111 s 88 ms			440 s 72 ms

Tableau 7.2 : Temps de compression parallèle pour l'image LENA (64, 128, et 256 Ko) sur de différents nombres de clients (2, 4 et 8 clients).

Image de Collie							
Nb client	N° client	Taille	Temps par Serveur	Temps par Client	Taille	Temps par Serveur	Temps par Client
2	0	256 x 256	**124 s 79 ms**	88 s 26 ms	512 x 256	**439 s 68 ms**	383 s 72 ms
	1			115 s 68 ms			430 s 34 ms
4	0		**60 s 96 ms**	52 s 90 ms		**223 s 76 ms**	215 s 69 ms
	1			52 s 62 ms			216 s 40 ms
	2			44 s 43 ms			191 s 91 ms
	3			52 s 78 ms			215 s 3 ms
8	0		**34 s 33 ms**	23 s 55 ms		**120 s 78 ms**	109 s 16 ms
	1			26 s 59 ms			109 s 41 ms
	2			22 s 14 ms			90 s 57 ms
	3			26 s 15 ms			108 s 4 ms
	4			28 s 2 ms			115 s 34 ms
	5			23 s 34 ms			96 s 1 ms
	6			26 s 20 ms			103 s 43 ms
	7			27 s 41 ms			111 s 93 ms
2	0	256 x 512	**440 s 39 ms**	383 s 76 ms	512 x 512	**1837 s 52 ms**	1529 s 12 ms
	1			431 s 55 ms			1829 s 34 ms
4	0		**223 s 55 ms**	215 s 47 ms		**1064 s 18 ms**	776 s 81 ms
	1			215 s 91 ms			950 s 21 ms
	2			191 s 85 ms			943 s 28 ms
	3			214 s 87 ms			1054 s 74 ms
8	0		**120 s 67 ms**	103 s 75 ms		**484 s 31 ms**	394 s 11 ms
	1			109 s 25 ms			459 s 73 ms
	2			90 s 52 ms			372 s 67 ms
	3			107 s 93 ms			371 s 84 ms
	4			115 s 23 ms			448 s 85 ms
	5			91 s 7 ms			476 s 81 ms
	6			103 s 64 ms			393 s 10 ms
	7			111 s 94 ms			434 s 68 ms

Tableau 7.3 : Temps de compression parallèle pour l'image Collie (64, 128, et 256 Ko) sur de différents nombres de clients (2, 4 et 8 clients).

Image de San Fransisco							
Nb client	N° client	Taille	Temps par Serveur	Temps par Client	Taille	Temps par Serveur	Temps par Client
2	0	256 x 256	**120 s 56 ms**	88 s 27 ms	512 x 256	**441 s 72 ms**	361 s 98 ms
	1			111 s 50 ms			433 s 3 ms
4	0		**60 s 96 ms**	53 s 22 ms		**223 s 71 ms**	217 s 17 ms
	1			52 s 62 ms			215 s 85 ms
	2			44 s 66 ms			182 s 7 ms
	3			52 s 62 ms			214 s 81 ms
8	0		**34 s 39 ms**	23 s 32 ms		**120 s 78 ms**	109 s 8 ms
	1			26 s 53 ms			109 s 35 ms
	2			22 s 8 ms			90 s 57 ms
	3			26 s 42 ms			107 s 93 ms
	4			27 s 84 ms			115 s 23 ms
	5			22 s 24 ms			95 s 90 ms
	6			24 s 99 ms			103 s 81 ms
	7			27 s 46 ms			111 s 50 ms
2	0	256 x 512	**441 s 99 ms**	433 s 15 ms	512 x 512	**1897 s 23 ms**	1787 s 82 ms
	1			361 s 85 ms			1888 s 18 ms
4	0		**223 s 27 ms**	215 s 64 ms		**1043 s 70 ms**	779 s 29 ms
	1			216 s 2 ms			893 s 86 ms
	2			191 s 85 ms			945 s 32 ms
	3			214 s 54 ms			1036 s 66 ms
8	0		**120 s 23 ms**	115 s 35 ms		**483 s 18 ms**	394 s 6 ms
	1			109 s 20 ms			454 s 51 ms
	2			90 s 47 ms			372 s 72 ms
	3			107 s 93 ms			371 s 79 ms
	4			104 s 68 ms			450 s 28 ms
	5			92 s 11 ms			476 s 81 ms
	6			103 s 70 ms			393 s 5 ms
	7			112 s 16 ms			434 s 24 ms

Tableau 7.4 : Temps de compression parallèle pour l'image San Fran (64, 128, et 256 Ko) sur de différents nombres de clients (2, 4 et 8 clients).

Les tableaux suivants contiennent les temps globaux de compression parallèle pour les trois images (LENA, COLLIE et SAN FRANSISCO) sur différents nombres de postes (2, 4 et 8).

Le tableau 7.5 contient les temps globaux (sur le serveur) de compression parallèle pour les images de Lena.

Le tableau 7.6 contient les temps globaux (sur le serveur) de compression parallèle pour les images de Collie.

Le tableau 7.7 contient les temps globaux (sur le serveur) de compression parallèle pour les images de San fransisco.

Ces trois tableaux ont la même forme qu'est la suivante:

La 1ère colonne concerne le nombre de postes clients pour chaque test.

La 2ème colonne concerne la taille de l'image sur laquelle ont été effectués les tests.

La 3ème colonne contient le temps global de compression parallèle sur le serveur pour un test bien déterminé (spécifique).

Les colonnes 4 et 5 sont identiques aux colonnes 2 et 3 et concernent les tests effectués sur d'autres images de taille différente.

Image de LENA				
Nb client	Taille	Temps de codage	Taille	Temps du Codage
2	256 x256	125 s 47 ms	512 x 256	469 s 28 ms
4		60 s 59 ms		223 s 55 ms
8		33 s 50 ms		120 s 18 ms
2	256 x 512	466 s 92 ms	512 x 512	1834 s 55 ms
4		223 s 93 ms		893 s 72 ms
8		120 s 7 ms		460 s 22 ms

Tableau 7.5 : Temps global de compression parallèle de l'image de LENA sur de différents nombres de postes (2, 4 et 8 postes).

Image de Collie				
Nb client	Taille	Temps deu codage	Taille	Temps du Codage
2	256 x256	124 s 79 ms	512 x 256	439 s 68 ms
4		60 s 96 ms		223 s 76 ms
8		34 s 33 ms		120 s 78 ms
2	256 x 512	440 s 39 ms	512 x 512	1837 s 52 ms
4		223 s 55 ms		1064 s 18 ms
8		120 s 67 ms		484 s 31 ms

Tableau 7.6 : Temps global de compression parallèle de l'image de COLLIE sur de différents nombres de postes (2, 4 et 8 postes).

Image de San Fransisco				
Nb client	Taille	Temps de codage	Taille	Temps du Codage
2	256 x256	120 s 56 ms	512 x 256	441 s 72 ms
4		60 s 96 ms		223 s 71 ms
8		34 s 39 ms		120 s 78 ms
2	256 x 512	441 s 99 ms	512 x 512	1897 s 23 ms
4		223 s 27 ms		1043 s 70 ms
8		120 s 23 ms		483 s 18 ms

Tableau 7.7 : Temps global de compression parallèle de l'image de SAN FRANSISCO sur de différents nombres de postes (2, 4 et 8 postes).

Les résultats montrés dans les tableaux plus hauts (tableau 7.1, tableau 7.5, tableau 7.6 et tableau 7.7) sont traduits sous forme de courbes dans les figures suivantes :

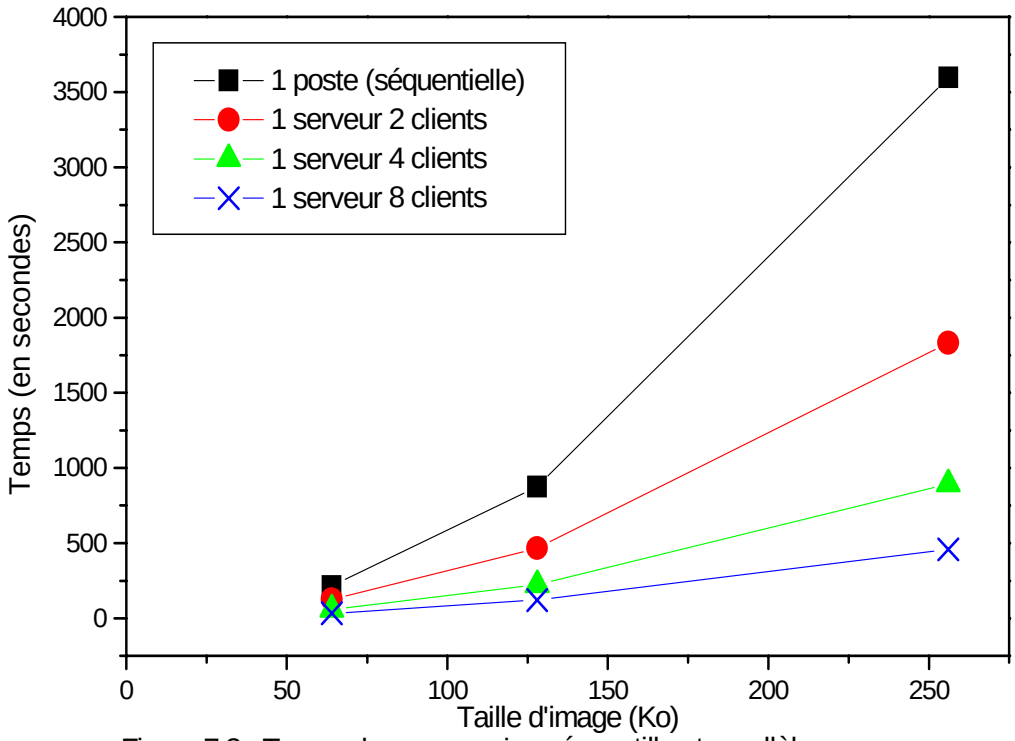

Figure 7.3 : Temps de compression séquentille et parallèle (2, 4 et 8 postes) sur l'image de LENA (64, 128 et 256 Ko).

Cette courbe montre les résultats obtenus après les tests effectués sur les images de Lena pour des différents nombres de postes: 1 poste (séquentiel), 2, 4 et 8 postes.

La courbe de la figure 7.4 montre les résultats obtenus après les tests effectués sur les images de Collie pour des différents nombres de postes: 1 poste (séquentiel), 2, 4 et 8 postes.

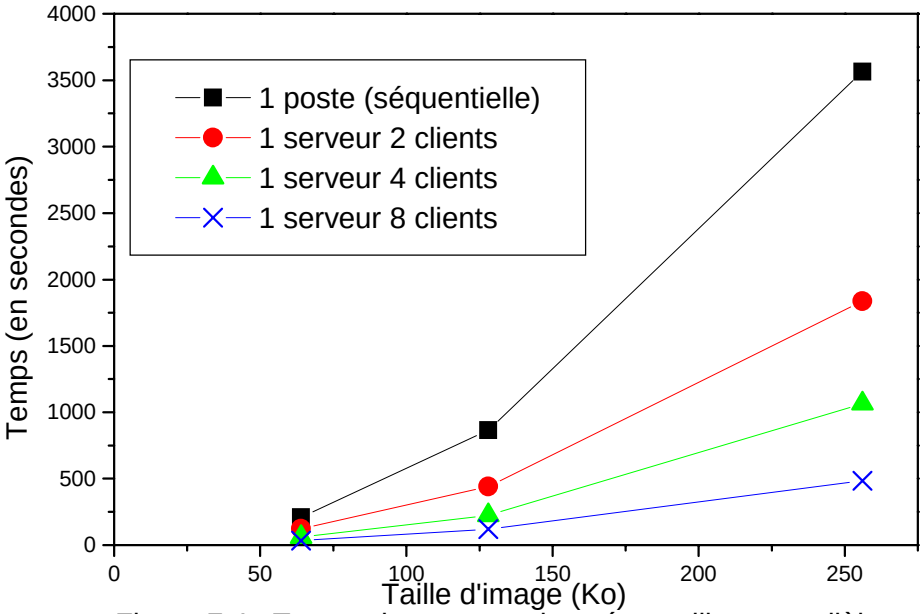

Figure 7.4 : Temps de compression séquentille et parallèle (2, 4 et 8 postes) sur l'image de Collie (64, 128 et 256 Ko).

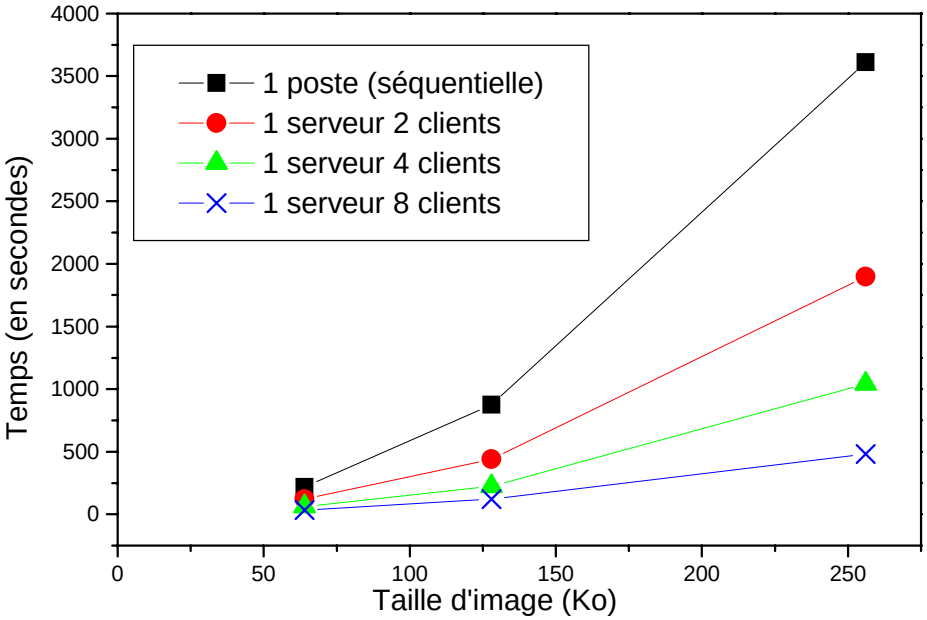

Figure 7.5 : Temps de compression séquentille et parallèle (2, 4 et 8 postes) sur l'image de Sanfran (64, 128 et 256 Ko)

Cette courbe montre les résultats obtenus après les tests effectués sur les images de San fransisco pour des différents nombres de postes: 1 poste (séquentiel), 2, 4 et 8 postes.

Les figures suivantes montrent des courbes concernant le temps de compression séquentielle et parallèle en négligeant le temps de communication pour les mêmes images et le même nombre de postes :

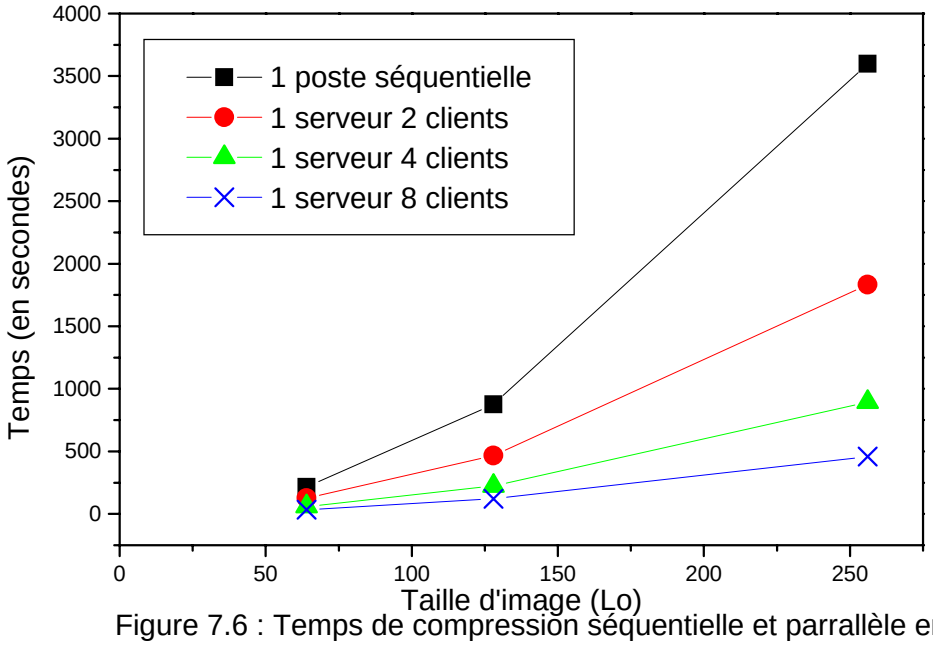

Figure 7.6 : Temps de compression séquentielle et parrallèle en négligeant le temps de communication pour les images de LENA

Cette courbe montre les résultats obtenus après les tests effectués sur les images de Lena pour de différents nombres de postes (2, 4 et 8 postes) en négligeant le temps de communication.

La courbe de la figure 7.7 montre les résultats obtenus après les tests effectués sur les images de Collie pour de différents nombres de postes (2, 4 et 8 postes) en négligeant le temps de communication.

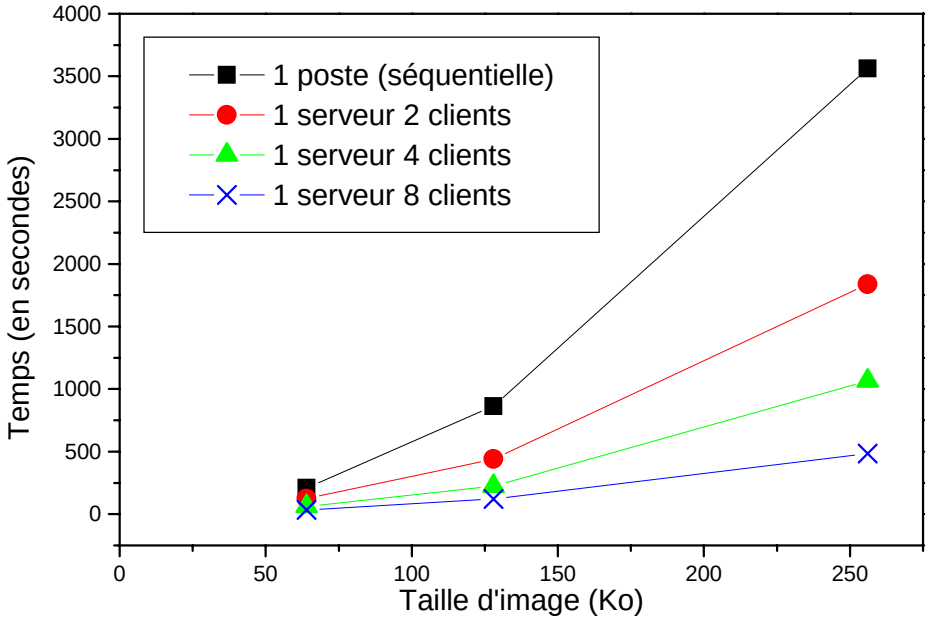

Figure 7.7 : Temps de compression séquentielle et parrallèle en négligeant le temps de communication pour les images de Collie

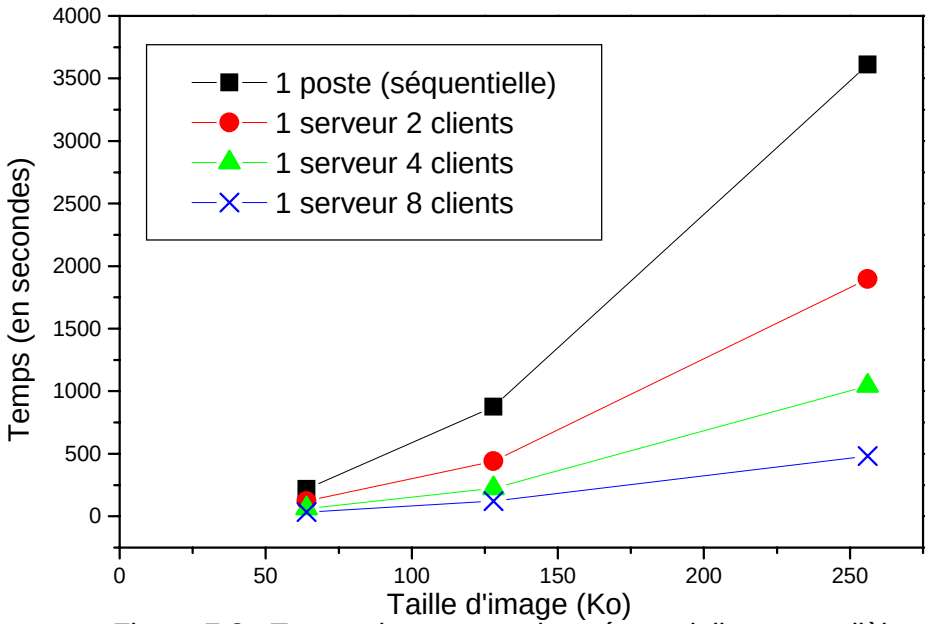

Figure 7.8 : Temps de compression séquentielle et parrallèle en négligeant le temps de communication pour les images de Sanfran

Cette courbe montre les résultats obtenus après les tests effectués sur les images de San fransisco pour de différents nombres de postes (2, 4 et 8 postes) en négligeant le temps de communication.

Nous avons testé notre application sur les images : LENA, COLLIE et SAN FRANSISCO de différentes tailles pour chacune (64 Ko, 128 Ko et 256 Ko). Les résultats obtenus pour les trois images sont montrés sous forme de courbes dans les figures : fig. 7.3, fig. 7.4 et fig. 7.5.

On remarque (fig. 7.3, fig. 7.4 et fig. 7.5) que le temps de codage en séquentiel est toujours supérieur à celui en parallèle dans tous les cas : 2, 4 et 8 postes clients et que les temps de compression en parallèle diminuent en augmentant le nombre de clients (esclaves). A l'aide d'une analyse détaillée des temps de compression on peut déduire que la communication et le système informatique sont la cause de la perte de performance. Le temps de communication inclus le temps d'attente d'envoi des données et des messages de confirmation de réception de données et le temps de transmission.

Nous remarquons aussi que le temps de calcul global est influencé par le temps de communication et le temps de calcul du client qui a terminé le dernier.
Théoriquement, les temps de compression dans chaque esclave sont égaux. Mais le système informatique réel a influencé sur le temps de calcul des différents clients. Les temps de transmission ne le sont pas car le maître (serveur) est séquentiel et ne répond qu'à un esclave à la fois et les autres doivent attendre leur tour. Les performances de compression s'améliorent en augmentant le nombre de clients (esclaves) parce que le temps de compression sur les nœuds clients diminue et que le temps de communication n'est pas important par rapport au temps de calcul sur chaque client. Mais si en augmente le nombre de clients d'une manière intéressante (ex : 32, 64, 128 et 256 clients) le temps de compression sur chaque client diminue mais le temps de communication reste considérable. Ce qui rend l'augmentation des performances médiocre surtout pour les images de petite taille (256 x 256) ou inférieure. Pour les images de grande taille (512 x 512) ou plus, le temps de communication est négligeable par rapport au temps de compression ce qui améliore les performances lorsque le nombre de clients est augmenté.

Le protocole TCP/IP est inadapté pour les applications de hautes performances sur réseaux de stations de travail parce qu'il n'est pas optimisé pour

la communication dans un réseau local. En effet, la communication entre deux postes est assurée, qu'ils soient dans la même salle ou dans deux continents différents, et ce de la même manière !. C'est la raison principale de l'émergence des nouveaux protocoles de communication spécialisée pour les réseaux de stations de travail.

Les performances crêtes correspondent au temps de compression en négligeant le temps perdus en communication (Overhead).

Dans notre cas si on néglige le temps de communication, les temps de compression seront tels que représentés dans les figures : fig. 7.6, fig. 7.7 et fig. 7.8.

On remarque que les performances obtenues de notre application en négligeant le temps de communication sont très proches des estimations théoriques :

TCP = TPS / NP

TCP : Temps de Compression Parallèle,

TPS : Temps de Compression Séquentielle,

NP : Nombre de Processeurs.

C'est à dire l'augmentation du nombre de clients nous permet de réduire le temps de codage et que le temps de compression par client est égal à celui en séquentiel divisé sur le nombre de postes.

L'utilisation des mécanismes de communication non adéquats (TCP/IP, Ethernet, …) a un mauvais impact sur les performances lorsqu'on utilise des images de petite taille avec un grand nombre de clients. L'utilisation des mécanismes de communication plus rapides et d'un système informatique fable et puissant améliorerait les performances de notre application.

Les tableaux 7.8, 7.9 et 7.10 sont consacrés aux accélérations de l'algorithme de compression en variant le nombre de postes clients. Pour chaque image, nous avons calculé les accélérations pour différents nombres de postes (1, 2, 4 et 8 postes).

Taille	256 x 256		256 x 512		512 x 512	
postes	Temps (s)	accélération	Temps (s)	accélération	Temps (s)	accélération
1	217	---	876	---	3600	---
2	125	1.73	467	1.87	1834	1.96
4	60	3.61	224	3.91	894	4.02
8	33	6.57	120	7.3	460	7.82

Tableau 7.8 : Compression de l'image de Léna de différentes tailles (accélération)

Taille	256 x 256		256 x 512		512 x 512	
postes	Temps (s)	accélération	Temps (s)	accélération	Temps (s)	accélération
1	211	---	865	---	3563	---
2	125	1.69	440	1.96	1837	1.94
4	61	3.46	223	3.88	1064	3.39
8	34	6.2	120	7.2	484	7.36

Tableau 7.9 : Compression de l'image de Collie de différentes tailles (accélération)

Taille	256 x 256		256 x 512		512 x 512	
postes	Temps (s)	accélération	Temps (s)	accélération	Temps (s)	accélération
1	217	---	876	---	3610	---
2	120	1.81	442	1.98	1897	1.9
4	61	3.55	223	3.93	1044	3.45
8	34	6.38	120	7.3	483	7.47

Tableau 7.10 : Compression de l'image de Sanfran de différentes tailles (accélération)

Les tableaux (7.8, 7.9 et 7.10) montrent que l'accélération croît lorsque le nombre de clients est augmenté. Ce qui prouve que l'accélération est pratiquement liée au nombre de stations de travail (clients) participant dans l'expérimentation.

Les résultats que nous avons obtenus montrent que les performances de l'algorithme s'améliorent quand les traitements en chaque nœud sont

indépendants ce qui justifie que la parallèlisation via les ranges est une méthode efficace.

Les performances de la machine physique interviennent naturellement dans les performances de l'ensemble.

7.3 Conclusion

Dans ce chapitre nous avons montré les résultats des tests que nous avons effectué en exécutant notre application sur un réseau Ethernet.

Les tests ont été effectués sur différentes images pour différents nombres de postes clients. Les résultats de ces derniers ont été presque identiques. Les différences entre les temps obtenus sur les clients sont dues principalement au système informatique utilisé.

Nous avons remarqué que la parallèlisation de l'algorithme de compression fractale via les ranges améliore considérablement la vitesse de codage.

Conclusion générale

Ce travail a été consacré à la compression fractale et plus particulièrement la parallèlisation de l'algorithme de compression à l'aide de cette méthode et son implémentation afin d'améliorer la vitesse de codage.

La compression fractale d'images fixes constitue une voie de recherche d'actualité. La représentation des formes fractales très complexes par des transformations IFS très simples constitue la base de cette méthode de compression. Cette méthode possède un potentiel lui permettant de figurer parmi les méthodes de compression efficaces. Le défaut majeur des techniques de compression fractale est leur lenteur dans la phase de codage.

Après une brève introduction à la théorie de l'image, nous avions parlé des principales méthodes de compression et nous avions présenté les bases théoriques de la compression fractale (IFS). Ensuite, nous avons montré comment les utiliser pour compresser les images. Cette méthode de compression consiste à coder les similarités entre blocs de l'image à l'aide de transformations affines contractantes. Avant de nous attaquer au sujet central de notre travail, nous avons présenté des notions sur le parallélisme et les architectures parallèles.

Dans ce travail, nous avions proposé une implémentation parallèle de l'algorithme via les ranges en utilisant une architecture MIMD à mémoire distribuée qui a été simulée par un réseau de stations de travail dont le modèle utilisé est 'client/serveur', en l'occurrence un réseau Ethernet dont la topologie est étoile. Les postes sont des machines AMD durron (1.3 GHz et 128 Mo de RAM) dotés du système windows 98.

Notre expérimentation a été effectuée sur plusieurs images de différentes tailles (Lena, Collie, San fansisco : 256x256, 256x512, 512x256 et 512x512 pour chaque image).

Les tests ont été effectués sur 2, 4 et 8 postes clients pour chaque image. Nous avons fait un prélèvement des temps de compression sur chaque poste ainsi que le temps global sur le serveur pour chaque test. Nous avions remarqué des différences légères entre les résultats obtenus. Ces différences sont dues au système informatique utilisé (logiciel et matériel).

Les mécanismes de communication non adéquats (TCP/IP, Ethernet, …) ont aussi influencé les résultats, ce qui nous ramène à dire que les performances de la

machine physique interviennent naturellement dans les performances de l'ensemble.

Les résultats ont été établis sous forme de tableaux détaillés puis une récapitulation des temps globaux a été faite sous forme d'un tableau pour chaque type d'images. Cette dernière a été interprétée par un ensemble de courbes.

Après une analyse des résultats, nous avions remarqué que le temps de codage en séquentiel est toujours supérieur à celui en parallèle et que les temps de compression en parallèle diminuent en augmentant le nombre de clients. L'accélération croît aussi à son tour ce qui prouve qu'elle est pratiquement liée au nombre de stations de travail (clients) participant dans l'expérimentation effectuée sur les différentes images.

Tout ça nous permet de dire que la parallèlisation améliore considérablement la vitesse de codage et qu'elle représente une solution efficace pour la résolution du problème de la vitesse de compression à l'aide des fractales qu'est le problème majeur de cette technique, et que l'accélération est pratiquement liée au nombre de stations de travail.

Comme suite à notre travail, nous proposons de tester notre application sur un réseau de station de travail qui utilise un vrai support de communication de hautes performances, des machines puissantes et un maître (serveur) multi-thread sur une machine multiprocesseur (bi-processeur ou quadri-processeur), pour atteindre des performances meilleures et plus proches de la performance crête.

Nous avons l'intention de développer notre application en lui ajoutant l'autre approche qui est la parallèlisation via les domaines et faire une comparaison entre cette dernière et l'approche que nous avions implémenté (parallèlisation via les ranges) pour montrer quelle est la meilleure du point de vue accélération de la vitesse et réduction du temps de codage.

Lexique

Partition R ……	partition de l'image composée de blocs destination;
Partition D ……	partition de l'image composée de blocs source;
R_i ……...	bloc destination numéro *i*. En anglais: *range block;*
D_i…….....	bloc source numéro *i*. En anglais: *domain block;*
b_2 ……..	bloc source décimé, superposé au bloc destination;
b_1 ……..	bloc constant, dont la luminance des pixels est égale à un;
collage ……..	transformation ramenant un bloc source sur un bloc destination: transformation spatiale + transformation dans l'espace des luminances de manière à ce que le bloc transformé $\hat{r}_n$ approxime le bloc destination r_n ;
transformation fractale	transformation d'une image A à partir de transformations élémentaires w_n composant l'opérateur W finalement contractant, de manière à ce que l'image W(A) approxime au mieux l'image A. Une transformation w_n opère le collage du bloc source $d_{\alpha(n)}$ sur le bloc destination r_n ;
MIMD …….	Architecture massivement parallèle: Multiple Instruction stream Multiple Data stream. Permet d'exécuter un programme différent sur tous les processeurs;
SIMD……	Architecture massivement parallèle: Single Instruction stream Multile Data stream. La même opération est appliquée à plusieurs données simultanément.
MISD ……	Architecture massivement parallèle: Multiple Instruction stream Single Data stream. Un seul flux de données reçoit plusieurs traitements simultanément.
Socket …….	Permet de créer une application pouvant communiquer avec d'autres systèmes par TCP/IP et ses protocoles associés;
Réseau Ethernet	Réseau reliant les postes point à point dont aucun poste n'a le contôle sur le réseau.

RLE ……..	Run Length Encoding: fait partie des méthodes de compression réversible.
LZ** …….	Algorithme de compression créé par Abraham Lempel et Jacob Ziv. Fait partie des méthodes de compression réversible.
LZW	Lempel Ziv Welch: extension de l'algorithme LZ.
DCT…….	Transformée en Cosinus Discret: fait partie des méthodes de compression irréversible.
JPEG…….. …….	Joint Photographie Experts Group: fait partie des méthodes de compression irréversible.

Annexe A

Les détailles des Algorithmes
LRC -- Huffman -- LZW

L'algorithme de RLC de compression

```
CodeRepet = 255
NbreRepet = 1 Lire(AncCaract)
Tant que pas fin fichier faire
    Lire (caractère)
    Si caractère = AncCaract Alors
        NbreRepet = NbreRepet+1
    Sinon
        Si NbreRepet > 3 Alors
            Ecrire(codeRepet)
            Ecrire(NbreRepet)
            Ecrire(AncCaract)
        Sinon
            Pour i :=l à NbreRepet faire
                Ecrire(AncCaract)
            Fin pour
        Fin Si
        NbreRepet =1
        AncCaract =caractère
    Fin Si
Fin tant que
```

L'algorithme de RLC de décompression

```
CodeRepet = 255 NbreRepet = 1
Tant que pas fin fichier faire
    Lire(caractère)
    Si caractère = CodeRepet Alors Lire(NbreRepet)
        Lire(CodeARepeter)
        Pour i :=1 à NbreRepet faire
            Ecrire(CodeARepeter)
        Fin pour
    Sinon
        Ecrire(caractère)
    Fin Si
Fin tant que
```

L'algorithme de Huffman de compression

```
/* construction de la table des fréquences FREQ
DIM =0      /* Dimension de la table
Lire(octet)
Tant que pas fin ensemble source Faire
    Recherche Ind   /* position de l'octet dans FREQ
    Si Ind = 0 Alors    /* nouvel élément = pas trouvé dans la table
        Insérer octet dans table FREQ > nouveau Ind
        DIM =DIM + 1
    Fin Si
    FREQ(Ind) = FREQ(Ind) + 1
    Lire(octet)
Fin Tant que
```

L'algorithme de Huffman de décompression

```
FREQ = vide
DIM =0
/* lecture de la table de décodage
Tant que pas de marque de fin de table Faire
    DIM =DIM + 1
    Lire(FREQ(DIM .octet,FREQ(DIM).Code)
Fin Tant que
/* construction de l'arbre à partir de la table
Lire(octet)
pointeur arbre = racine
Tant que pas fin de fichier Faire
    Tant que feuille et i < 8 Faire
        Si octet[i] = 0 Alors
            Branche droite
        Sinon
            Branche gauche
        Si feuille = caractère Alors
            Récupère(car)
        i=i+1
    Fin Tant que
    Si i = 8 Alors
        Lire(octet)
    Sinon
        Ecrire(car)
        Pointeur arbre = racine
    Fin Si
Fin Tant que
```

L'algorithme de LZW de compression

```
/* Initialisation
DICO = table ASCII   /*dictionnaire dynamique
LATENT = "
MAXDIM = 1024  /* taille maxi du dictionnaire
NBCAR = 257  /* nbre de cara. Courant dans DICO
/* Boucle de traitement
Lire(C) /* lecture du premier caractère
Tant que C <> EOF Faire    /* test de fin de caractères
    Si LATENT = " Alors
        LATENT = C
        ANCRANG = ASCII(C)   /* sauve le range de C
    Sinon
        CHAINE = LATENT + C   /* Concaténation
        /* Recherche dichot. Ou dictionnaire utilisant le hash coding, renvoie le rang où
        devrait se trouver la chaîne
        TROUVE = RECHERCHE (CHAINE, DICO, RANG)
        Si TROUVE Alors
            LATENTE = CHAINE; ANCRANG = RANG
        Sinon
            EMETTRE (ANCRANG)
            AJOUTER (CHAINE, DICO, RANG)
            NBCAR = NBCAR + 1
            Si NBCAR = MAXDIM Alors
                EMEITRE (257) /* remise à zéro DICO
                NBCAR = 257
            Fin Si
            LATENTE = C; ANCRANG = ASCII (C)
        Fin Si
    Fin si
    Lire (C)   /* lecture du caractère courant
Fin Tant que
EMETTRE(ANCRANG, 256)   /* indice dernière chaîne
```

L'algorithme de LZW de décompression

```
/* initialisation
DICO = table ASCII /* ensemble de chaîne construites
LATENT = ''
FININFO = 256 /* code indiquant la fin de compression
/* Boucle de
traitement
RECEVOIR(CODE
)
Tant que CODE <> FININFO Faire
    Si LATENT = '' Alors
            LATENT = CODE
            ECRIRE(CODE)
    Sinon
        Si CODE< 257 Alors
            /* Evaluation d'un chaîne qui ne figure pas dans le dictionnaire
            ECRIRE (CODE)
            CHAINE = LATENT +
           CODE AJOUTER(CHAINE,
           DICO) LATENT = CODE
        Sinon
            CHAINE = RECHERCHERBIS(CODE, DICO)
            ECRIRE(CHAINE)
            TEMPO = LATENT + CHAINE
            [1]    AJOUTER(TEMPO, DICO)
            LATENT = CHAINE
        Fin Si
    Fin Si
   RECEVOIRE (CODE)
Fin Tant que
```

Annexe B

Les Détailles des Algorithmes de Compression Fractale

Algorithme de Partitionnement OuadTree Pour les Blocs de 32×32

```
Procedure Quadtree (cornerX, corerY, Size : entier)
Var
    O_Opt , S Opt , RMS : réel
    X_Opt ,Y_Opt, R_Opt : entier
DEBUT
    Si Size > 32 Alors
        début
          Quadtree(ComerX, cornerY,size Div2)
           Quadtree(comerX+Size Div2, comerY, Size Div2)
          Quadtree (comerX, cornerY+size Div2, Div2)
          Quadtree(ComerX+ Size Div2,comerY+ Size Div2, Size Div2)
        fin
    Sinon
        début
        Trait_Bloc(ComerX,ComerY,Size,O_Opt,S_Opt,RMS,X_Opt,Y_Opt,R_Opt)
          Si (RMS > Threshod) et (Size > 4) Alors
            début
                QuadTree(CornerX,ComerY,Size Div 2)
                QuadTree(ComerX + Size Div 2,ComerY,Size Div 2)
                QuadTree(CornerX,ComerY+ Size Div 2,Size Div 2)
                QuadTree(ComerX + Size Div 2,CornerY +Size Div 2,Size Div 2)
            fin
          Sinon
            début
                Store(O_Opt,S_Opt,R_Opt,X_Opt,Y_Opt,Size)
            fin
        fin
FIN.
```

Algorithme de construction de la Bibliothèque des Domaines

```
DEBUT
    Pour i:=1 à 5 Faire SIZE[il := 0;
    Z:=4; t:=1;
    Tant que (t<=4) Faire
        début
            Z:=Z*2; i=1 ; k:=l ;
            Tant que (i<=XSize-Z) Faire
                début
                    J:=1;
                    Tant que (i<=YSize-Z) Faire
                        début
                            Lib[t,l].x:=j;
                            Lib[t,l].y:=i ;
                            Lib[t,l].Sai:=calc_ai(i,j) ;
                            Lib[t,l].Sai2 :=calc_ai2(i,j) ;
                            Inc (size[t]) ;
                            J:=j+z div 2 ;
                        fin;
                    I:=i+z div 2 ;
                fin;
                        T :=t+1
        fin;
FIN;
```

Z : taille du bloc domaine.

Size : Un tableau qui contient pour chaque type de bloc leur nombre total.

Xsize,Ysize : largeur et hauteur de l'image .

T : représente la taille du bloc de balayage.

Calc_ai , calc_ai2 : deux fonctions qui calculent la somme et la somme des carrés des éléments d'un bloc.

Algorithme de Classification des Blocs

```
Procedure Classer_bloc (cx, cy, T:entire ; Var  id1, id2 : byte );
Var a1, a2: Tableau [1..4] de reels; Ord : Tableau [1..4] de byte;
DEBUT
Calcul(a1[1], a2[1], cx, cy, T div 2);
Calcul(a1[2], a2[2], cx, cy + T div 2, T div 2);
Calcul(a1[3], a2[3], cx + T div 2, cy, T div 2);
Calcul (al[4],a2[4],cx+T div 2, cy+ T div 2 ,T div 2 );
Pour i:= 0 à 3 Faire Ord[i+l] :=i ;
Pour i := 1 à 3 Faire
   début
      Si al [i] < al [i+l] Alors
         début
            Permute(al[i],al[i+l]) ; Permute(ord[i],ord[i+1]) ;
         fin ;
   fin ;
I:=0;
Pour  i:= 0 à 3  Faire
    Ord[i]:= (ord[i] - ord[1] mod 4+4) mod 4 ;
Répéter I :=i+l;  Jusqu'à ce que (ord[i] =2) ;
Id1:=i;
Pour i:= 0 à 3 Faire ord[i+l]:=i;
Id2 := 0;
Pour  i:= 2 à 0 Faire
    Pour  j:= 0 à i Faire
        début
            Si (ord[j] >ord[j+l]) Alors Permute (ord[j] >ord[j+1])
            Si  (ord[j] =0) ou  (ord[j+l] =0) Alors id2:=id2+6 sinon
            Si  (ord[j] =1) ou  (ord[j+l] =1) Alors id2:=id2+2 sinon
            Si  (ord[j] =2) ou (ord[j+1] = 2 )  Alors id2:=id2+1 ;
        fin ;
FIN ;
```

Procédure de Classification des Blocs Domaines

```
Procedure Classification ;
Var I,J,idl,id2 : entier ;
 DEBUT
      Pour i :=1 à 4 Faire
      Pour j :=1 à Size[i] Faire
          début
               Classer_ bloc (lib [i j].x , lib[i,j].y, , 2^i, idl,id2) ;
               lib[i,j].class_esp := id1 ;
               lib[i, j].class_ver := id2 ;
          fin ;
FIN ;
```

Bibliographie

[1] M. Barakat, Partitions arborescentes et compression fractale, Thèse, Institut National, Polytechnique Toulouse, France, Janvier 2000.

[2] R.D. Boss, E.W. Jacobs, "Fractal-Based Image Compression", NOSC Technical Report 1315, September 1989. Naval Ocean Systems Center, San Diego CA 92152-5000.

[3] R.D. Boss, E.W. Jacobs, "Fractal-Based Image Compression II", NOSC Technical Report 1362, June 1990. Naval Ocean Systems Center, San Diego CA 92152-5000.

[4] A. Jacquin, A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding, Doctoral Thesis, Georgia Institute of Technology, 1989.

[5] Y. Fisher, "Fractal Image Compression", Institute for Non-Linear Science 0402, University of California, San Diego La Jolla, CA 92093, SIGGRAPH 92 Course Notes.

[6] C. Jean, "Compression fractale d'images", L'ULB 4 octobre 2001.

[7] Encyclopédie Microsoft® Encarta® 2002. © 1993-2001 Microsoft Corporation, Cd-Rom.

[8] http://www.kaddour.com/

[9] C. Diltsm, Numérisez vos collections, RCIP Octobre 2001. http://www.chin.gc.ca/Francais/Contenu_Numerique/Guide_Gestionnaies/

[10] "http://membres.lycos.fr/compressions/menua.html", Octobre 2002.

[11] P. Faust, C. Guy, D. Passagne, I. Siffert, P. Silvestre, D. Zeller sont des professeurs de Technologie de l'Académie de Strasbourg.

[12] C. Lepec, Les Graphiques Sur Ordinateurs, N. RIMOUX & A. COLIN Editeurs. Paris, 1991.

[13] C. Boudry. URFIST de Paris/Ecole des Chartes
http://www.ccr.jussieu.fr/urfist/image_numerique/home_image.htm, Octobre 2002.

[14] F. Davoine, Compression d'Images par Fractales Basée sur la Triangulation de Delaunay, Thèse l'INPG, Institut National Polytechnique de Grenoble, Décembre 1995.

[15] R. Souadnia & K. Benmahamed, "Compression des Images Numériques Fixes Par les Fractales". Proceeding FRACTALES' 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[16] A. Ali-Pacha & N. Hadj-Said, "Compression Des Images Fixes Par Fractale: Partitionnement Quadtree". Proceeding FRACTALES 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[17] R. Sylvain, "La Compression de Données", Club Photoshop de Nantes. Conférence du 14 octobre 1999.

[18] C. Benoit & A. Dusson, "La compression de données informatiques". Juin 1999. http://www.esil.univ-mrs.fr/~cbenoit/projets/comp/

[19] J. F. Pillou, "Comment ça marche? L'informatique – Compression d'images", Année 1998. http://www.commentcamarche.net/video/

[20] F. Davoine, J.-M. Chassery, "IFS et applications en traitement d'images", chapitre du livre "Lois d'échelle, fractales et ondelettes" (P. Abri, P. Gonçalvès, J. Lévy-Véhel Eds.), Hermes Science Publications. 2002.

[21] A. Ameziane, Compression d'Images par Fractale, Thèse magister, Faculté des sciences de l'ingénieur, Département d'Informatique, Université de Batna, Année 2002.

[22] P. Flandrin, P. Abry et P. Gonçalvès, "Analyses en ondelettes et lois d'échelle", Dans Fractals et Lois d'Echelle, taite IC2 (Hermes Sciences Publications, Ed.). 2002.

[23] L. Mahiddine, K. Achour, M. Benkhelif, "Segmentation de Texture par Caractérisation Fractale", Proceeding FRACTALES' 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[24] Larousse dictionnaire de la langue française, édition 2000.

[25] Le Jargon Français v 3.3.70-09/08/2002.

[26] M.A.Aitouche & M.Djeddi, "Génération d'Images de Synthèse Par les Fractales et les Systèmes de Fonctions Itérées", Proceeding FRACTALES' 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[27] J. P. Louvet, "les fractales", http://fractals.iut.u-bordeaux1.fr/default.htm.

[28] M. K. Kholladi, "Les Structures Fractales des Objets Urbains dans un SIG", Proceeding FRACTALES' 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[29] J. Lévy Véhel et C. Tricot, "Analyse fractale et multifractale en traitement du signal", Année 2000.
http://www.inrialpes.fr/is2/people/pgoncalv/IC2/

[30] L. Hamami & N. Lassouaoui & L. Mahiddine, "Différentes Méthodes d'Estimation des Paramètres Fractals: Application aux Images", Proceeding FRACTALES' 2000, Université Mentouri Constantine, Algérie, Novembre 2000.

[31] E. Lutton. Algorithmes génétiques et fractales. Rapport d´habilitation Université de Paris XI Orsay, 11 Février 1999.

[32] J. Istas et A. Benassi, " Processus Auto-Similaires ", année 2000.
http://www.inrialpes.fr/is2/people/pgoncalv/IC2/

[33] B. Peroche & D. Ghazanfarpour & D. Michelucci & M. Roelens, Informatique Graphique méthodes et modèles, $2^{ème}$ édition, Editions HERMES. Paris, 1998.

[34] F. Cappello, "Une introduction aux architectures parallèles", LRI, Université Paris-Sud, 91405, Orsay, France. 5 octobre 1998.

[35] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye. "The IBM RISC System/6000 processor: Hardware overview". IBM Journal of Research and Development, 34(1):12-22, January 1990.

[36] Bernstein. "Analysis of programs for parallel processing". IEEE Transactions on Electronic Computers, pages 757-762, 1966.

[37] Thinking Machines Corporation, Cambridge, MA. The Connection Machine CM5 Technical Summary, October 1991.

[38] George S. Almasi and Allan Gottlieb. Highly Parallel Computing. Benjamin/Cummings division of Addison Wesley Inc., 1989.

[39] C. Hufnagl et A. Uhl, "Algorithms for Fractal Image Compression On Massively Parallel SIMD Arrays", PCS' 97 International Picture Coding Symposium, Berlin, Germany 1997.

[40] Ch. Hufnagl, J. Hämmerle, A. Pommer, A. Uhl and M. Vajtersic. "Fractal Image Compression On Massively Parallel Arrays". RIST++, Salzburg University, Austria 1999.

[41] K. Shen, G. W. Cook, L. H. Jamieson et E. J. Delp, "An Overview Of Parallel Processing Approaches to Image and Video Compression", Proceedings Of The SPIE Conference on Image and Video Compression, Volume 2186, February 1994, San Jose, California, PP. 197-208.

[42] A. Uhl, and J. Hämmerle, "Fractal Image Compression on MIMD architectures I: Basic Algorithms", RIST++, Salzburg University, Austria. Melbourne, Februray 1996.

[43] P.Latu, *Technologie Ethernet*, Linux France, Juin 2002, Home page www.linux-france.org.

[44] R.Chéramy, *Réseaux IP Un réseau local : Ethernet*, Ecole Centrale Paris, France, Février 2003, Home page http://robert.cheramy.net/ecp/.

[45] A. Uhl et J. Hammerle, "Issues in implementing block-based image compression techniques on parallel MIMD architectures", in J. Biemond and E.J. Delp, editors, Visual Communications and Image Processing '97, volume 3024 of SPIE Proceedings, pages 494-501, San Jose, Fevrier 1997.

Résumé

La compression fractale fait partie des méthodes de compression irréversible. Elle est fondée sur la théorie des systèmes de fonctions itérées (IFS). Divers travaux ont montré que cette méthode possède un potentiel lui permettant de figurer parmi les méthodes de compression efficaces. Le défaut majeur des techniques de compression fractale est leur lenteur à la phase de codage. La compression d'une image de taille moyenne à l'aide de cette méthode peut prendre quelques heures sur un ordinateur personnel.

Divers travaux de recherches ont été consacrés pour l'amélioration de la vitesse de compression tel que la classification des blocs et l'intégration du parallélisme dans l'algorithme de compression fractale.

C'est dans ce contexte que s'inscrit notre travail. Nous essayons dans ce mémoire d'accélérer la vitesse de compression de l'algorithme. L'existence de la propriété de parallélisme dans l'algorithme de compression fractale nous a permis de proposer une implémentation parallèle.

Les résultats que nous avons obtenus ont prouvé que l'intégration du parallélisme dans l'algorithme permet d'accélérer la vitesse de codage d'une manière importante.

Mots clés : Compression d'images, Architecture MIMD, Algorithme parallèle, Compression fractale d’images.

Abstract

Fractal compression belongs to the class of irreversible image compression methods. It is based on the theory of iterated function systems (IFS). Several works showed its efficiency. The defect major of techniques of fractal compression is their slowness to the phase of coding. The compression of an average size image with the help of this method can take some hours on a personal computer.

Various works of researches have been devoted for the improvement of the speed of compression such that the classification of blocks and the integration of the parallelism in the algorithm of fractal compression.

It is in this context that enters our work. We try in this memory to accelerate the speed of compression of the algorithm. The existence of the property of parallelism in the algorithm of fractal compression has allowed us to propose a parallel implementation.

Results that we have obtained have proven that the integration of the parallelism in the algorithm allows to accelerate considerably the speed of coding.

Keywords: image compression, MIMD architecture, parallel algorithm, fractal image compression.