Olivier GUILLOT
Jean LAURENT

E.N.S.E.R.B. 2000
3^{me} année Informatique

Parallélisation d'un compresseur MPEG-II

Responsable : Monsieur F.PELLEGRINI

Table des matières

Conclusion **3**

1 Le format MPEG II **4**
1.1 Conversion RGB-YUV 4
1.1.1 Sous-échantillonnage des composantes de chrominances 4
1.2 Compensation de mouvements 5
1.3 Les trois types de trames 6

2 Transformation du programme séquentiel en bibliothèque **8**
2.1 L'existent 8
2.2 L'interface 9
2.3 Approche du code 9
2.4 Le passage des paramètres 10

3 La parallélisation **12**
3.1 Première parallélisation 12
3.1.1 Conversion RGB-YUV 12
3.1.2 Répartion des données 12
3.1.3 Compensation de mouvement 14
3.1.4 Prédiction d'erreur 14
3.1.5 Ecriture du flux 14
3.2 Optimisation de la parallélisation 14

4 Régulation dynamique de charge **16**

5 L'architecture du projet **18**

6 Extensibilité **20**
6.1 Charge en calcul 20
6.2 Coût en communication 20

7 Utilisation de la bibliothèque pm2v **23**
7.1 installation 23
7.2 Interface 23
7.3 pm2v_open() 23
7.4 pm2v_add_image() 24
7.5 pm2v_close() 24

8 Les résultats **25**
8.1 IBM SP2 25
8.2 Sun enterprise 450 25
8.3 Réseau hétérogène 27
8.4 Réseau de stations de travails 27

9 Améliorations possible **29**
9.1 putpict() 29
9.2 le réseau 29

Conclusion **31**

Bibliographie **33**

Introduction

Le principal problème dans la transmission et le stockage sous forme numérique de vidéos est leurs taille (Un film de 2h qualité télévision prend 80 Giga Octets).
Dans ce cadre différentes méthodes de compressions ont été développées afin de réduire leurs tailles. Ces méthodes sont relativement efficaces pour le facteur de compression, mais les quantités énormes de données à traiter (le plus souvent de façon statistiques) impliquent un temps considérable de calcul.

Pour réduire ce temps de traitement à une valeur raisonnable, une approche classique est de faire appel à la parallélisation. Les données à traiter sont découpées en morceaux et affectées chacune à des processeurs différents.

Pour expérimenter cette approche nous avons donc pris un encodeur du domaine public (mpeg2encode v1.2a) écrit par le **MPEG II Software Simulation Group** dans l'objectif de le rendre parallèle.

Chapitre 1

Le format MPEG II

Il existe deux grandes classes de compressions vidéos.

– Compression sans perte.

– Compression avec perte.

Les méthodes de compressions avec pertes s'appuient sur la physiologie de la vision humaine. En effet il est inutile d'encoder des informations qui de toutes façon ne seront pas visibles par l'œuil humain, c'est justement cette méthode qu'utilise les formats MPEG.

La compression MPEG II (Motion Picture Expert Group) est un format de compression vidéo agrée par l'ISO (ISO/IEC 1318-2). [1]
Cette compression repose essentiellement sur deux types de compression:

– Compression spatiale des images.

– Réduction des redondances temporelles.

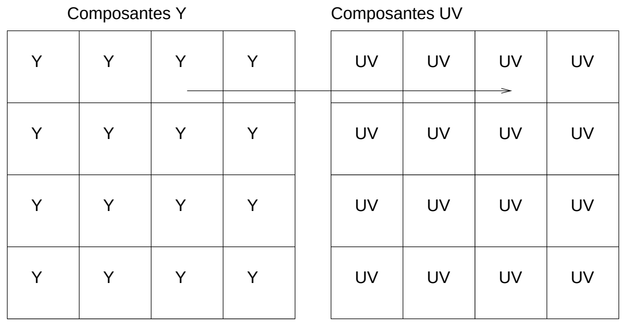
1.1 Conversion RGB-YUV

MPEG ne traite que de la vidéo RGB, ils font donc convertir l'entrée dans cette base.
Dans le format YUV, les composantes de luminance (Y) et de chrominances (U et V) sont séparées.
La conversion du format RGB au format YUV s'effectue par une multiplication d'un vecteur par une matrice 3x3.

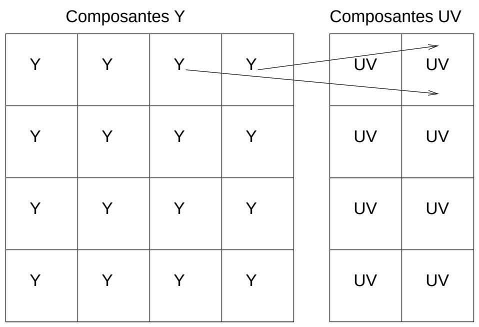
1.1.1 Sous-échantillonnage des composantes de chrominances

L'œuil humain est beaucoup plus sensible à la luminance qu'aux composantes de chrominances.
Donc en stockant moins d'informations sur la chrominance, la qualité de l'image reste sensiblement la même tout en gagnant de la place.
Différents formats de stockages sont couramment utilisés:

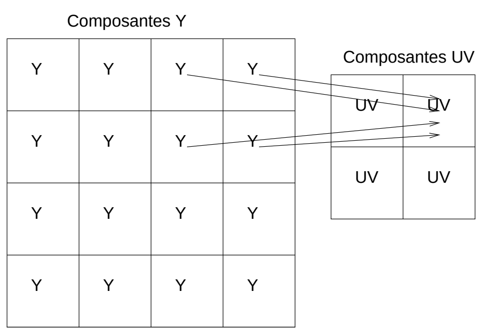
– Format 444: les 3 plans (YUV) sont stockés tels quels sans sous échantillonnage.



– Format 422: Un doublet de chrominance (UV) est commun à deux informations de luminances.



– Format 420: Un doublet de chrominance (UV) est commun à quatres informations de luminances.



1.2 Compensation de mouvements

Le principal gain en compression de MPEG réside dans la **compensation de mouvements**. Cette technique est basé sur la cohérence temporelle de la vidéo à compresser. En général tous les éléments d'une image à la suivante restent les mêmes mais sont légèrement déplacés. Ces mouvements peuvent être représentés à l'aide de vecteurs et une prédiction d'erreurs. Dans le cas de MPEG, chaque image est découpée en carrés de 16x16 appelés **macroblocs**. L'algorithme recherche dans l'image précédente la portion d'image ressemblant le plus au macrobloc courant, puis il soustrait les deux et encode le résultat à l'aide d'une **DCT** (Discret Cosine Transform). L'algorithme essaye donc de compenser les mouvements éventuels en retrouvant la portion d'image dans l'image précédente et en codant les différences éventuelles entre les deux.

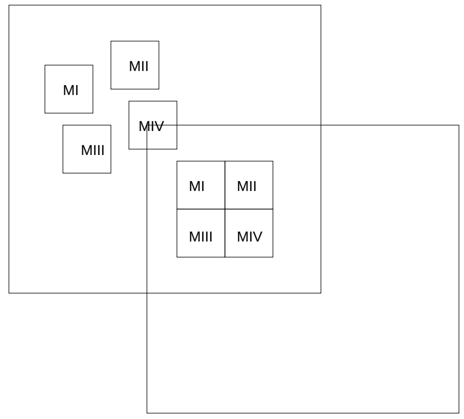


FIG. 1.1 - *Macroblocs*

1.3 Les trois types de trames

Les compressions MPEG I et II distinguent essentiellement 3 types de trames dans le flux codé:

- les I-pictures pour intra coded pictures sont codées sans références aux autres images.Elles constituent des points d'acces à la séquence codées où l'ont peut commencer à décoder.Elles sont codées avec une compression modérée.
- les P-pictures pour predictive coded pictures sont mieux compressées puisqu'elles utilisent la prédiction par compensation de mouvement par rapport à la plus proche image I ou P précédente. Elles servent de références pour les prédictions à venir.
- les B-pictures pour Bidirectionnally-predictive coded pictures sont les images les plus compressées puisqu'elles requièrent des images précédentes et suivantes comme référence pour la compensation de mouvement. Ces images sont des interpolations (au sens de la compensation de mouvement) entre deux images I ou P

La norme MPEG II laisse libre le choix de la séquence d'encodage. L'utilisateur peux donc la choisir librement, par exemple la séquence IIIII est valide.
Il est à noté que l'introduction excessive de trames B introduit un retard considérable dans le codage. En effet pour la sequence IBBBBPBBBBB, le codeur doit avoir les six premières images avant de pouvoir coder la deuxième.
Une telle séquence est appelé **GOP** (Group of picture).

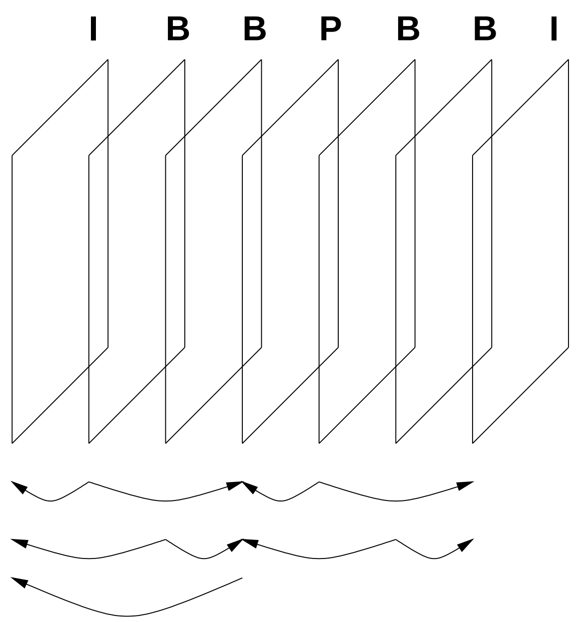


FIG. 1.2 - *Macroblocs*

Chapitre 2

Transformation du programme séquentiel en bibliothèque

2.1 L'existent

L'encodeur dont nous partons prend des fichiers images au format PPM, YUV ou trois fichiers par images représentants chacun un des tableaux Y,U et V. en sortie il génère un fichier mpeg2(.m2v) et un fichier de statistique. Il est très peu performant puisqu'il s'agit d'un programme se contentant de suivre les normes MPEG sans la moindre optimisation. Il est à noter que le code fourni ne respectait pas les plus simple règles de programmation(utilisation d'une centaine de variables globales à tout le programme, quasiment aucun commentaires, etc...) Les paramètres lui sont fournis dans un fichier. Une fois les paramètres analysées le programme effectue un appel à la fonction **putseq()** par image à coder. Voici une version simplifiée de **putseq()**

```
void putseq(void){
  readframe();
  motion_estimation();
  predict();
  dct_type_estimation();
  transform();
  putpict();
  boucle sur tous les macroblocks
    iquant_intra();
  itransform();
}

readframe(nom de l'image courante){
  lit le fichier d'image et convertit l'image en YUV au format
  employé(4:2:0,4:2:2 ou 4:4:4)
  l'image est stockée dans un buffer global.
}

putpict(){
  écrit l'image codée sur le disque dur en respectant un code de
  Huffman prédéterminé. C'est ici que l'on vérifie la taille de
  l'image produite car on doit assurer un certain débit maximal
```

```
}
```

predict(), **dct_type_estimation()** et **transform()** effectuent toute la compensation de mouvement.

Evidement **putseq()** lit les images dans l'ordre où il les traite, c'est à dire pour une séquence IBBP dans l'ordre IPBB puisqu'il a besoin de l'image P pour encoder les images B. Or notre bibliothèque prend en entrée les images dans l'ordre d'affichage, il nous a donc fallut crée un tampon dans lequel on accumule les images jusqu'a former un **GOP** entier. Ensuite , seulement, on peut lancer les calculs sur celui-ci.

2.2 L'interface

L'interface de notre bibliothèque doit être simple et permettre de traiter un flux d'images en entrée c'est pourquoi nous avons opté pour l'interface suivante:

- int pm2v_open(pm2v_stream* stream, char * param,FILE * output,pm2v_communicator com,int master_id,int my_id,int nb_proc);
- void pm2v_add_image(pm2v_stream* stream,unsigned char* img);
- void pm2v_close(pm2v_stream* stream);

Où l'utilisateur doit déclarer une variable de type pm2v_stream, qui contient les variables nécessaires tout au long de l'encodage. Ainsi tous les appels à notre bibliothèque prennent comme premier argument un pointeur sur pm2v_stream.

2.3 Approche du code

Nous avons commencé par "nettoyer" le code, c'est à dire éliminer les variables globales. Ce travail effectué nous partons sur un code thread-safe bien plus lisible puisque chaque effet de bord nous est signalé par un acces à la structure pm2v_stream. Sans ce travail préalable, il aurait été impossible de discerner quelles variables devaient être échangées entre les processeurs. Après une étude approfondie du code nettoyé nous avons analysé le temps passé par l'encodeur séquentiel dans chaque sous-fonction de putseq() de manière à voir les fonctions à paralléliser en priorité. Voici la répartition du temps machine pour chacune des sous fonctions de putseq() :

readframe()	10,9%
motion_estimation()	46,4%
transform()	18,9%
putpict()	11%

L'étude des effets de bord nous montre que les seules variables à mettre à jour lors de la compensation de mouvement sont la structure mbinfo et le tableau de bloc blocks.

short (*blocks)[64];

```
struct mbinfo {
  int mb_type; /* intra/forward/backward/interpolated */
  int motion_type; /* frame/field/16x8/dual_prime */
  int dct_type; /* field/frame DCT */
```

```
  int mquant; /* quantization parameter */
  int cbp; /* coded block pattern */
  int skipped; /* skipped macroblock */
  int MV[2][2][2]; /* motion vectors */
  int mv_field_sel[2][2]; /* motion vertical field select */
  int dmvector[2]; /* dual prime vectors */
  double act; /* activity measure */
};
```

Le code originale gérant un tableau global de ce type:
struct mbinfo mbi;

2.4 Le passage des paramètres

Nous avons du remplacer le mécanisme originel de passage des paramètres de compression. En effet à l'origine ceux-ci étaient lues à partir d'un fichier. Cette méthode n'étant évidemment pas pertinente pour une bibliothèque, nous l'avons donc remplacé par le passage d'une chaîne de caractère décrivant tous les paramètres.
La solution de la chaîne de caractère a été retenu car elle permet de masquer la structure vis-à-vis de l'utilisateur.

Nous avions besoin de passer des valeurs numériques, des chaînes de caractères ainsi que des tableaux à l'encodeur. Nous avons décidé d'utiliser lex et yacc bien que le grammaire de notre langage soit très simple.

La chaîne paramètre est constituée d'une suite d'affectations de la forme:

affectation ⇒ *tableau*;
| *expression*;

tableau ⇒ {*tableau*};
| {*tableau*,*liste_tableau*};
| expression;

liste_tableau ⇒ *tableau*,*liste_tableau*;
| *tableau*;

expression ⇒ (expression) **;**
| +*expression* | -*expression*;
| *expression*+*expression*;
| *expression*-*expression*;
| *expression*×*expression*;
| *expression*/*expression*;
| **entier** | **entier;**

En réalité, le programme n'accepte que des tableaux de dimensions maximale égale à deux. De plus dans le cas des tableaux à deux dimensions, les sous-tableaux doivent être de taille identique.

Le programme possède une liste de tous les symboles valides dans la structure **paramParse**. Cette structure contient en plus pour chaque symbole, son type (entier,chaîne...) et un pointeur sur la variable à modifier.

A l'initialisation, le programme lit une chaîne contenant les paramètres par défaut de l'encodeur, ainsi l'utilisateur n'a besoin que de changer les valeurs l'intéressant.

A l'exécution, le pogramme recherche pour chaque affectation grammaticalement valide si le symbole existe, si ce n'est pas le cas, la fonction termine et la bibliothèque signale l'erreur.

Globalement les symboles reprennent les champs du fichier de configuration originel, à l'exception **motion_vectors_parameters** qui regroupe les paramètres pour les trames P et B.

Chapitre 3

La parallélisation

Afin d'avoir un délai d'encodage faible, nous avons choisi de paralléliser l'encodage au niveau trame et non pas au niveau GOP.
L'unité logique naturelle pour la parallélisation semble donc être le macrobloc, en effet c'est une unité atomique dans le principe de la compression MPEG II.

Notre solution retient donc un parallélisme à grain moyen, en effet une image vidéo qualité télévision nécessite environ 1500 macroblocs.

3.1 Première parallélisation

Après analyse du code, nous avons déterminer les fonctions qui étaient intrinsèquement parallèles. Nous avons aboutit à une première version de l'encodeur (fig 3.1).

3.1.1 Conversion RGB-YUV

readframe() ne joue plus son rôle pour ce qui est de lire le fichier puisque nous passons désormais un tableau de caractères à la fonction pm2v_add_image(), par contre elle continue d'effectuer la conversion RGB vers YUV. C'était cette conversion qui prenait le plus de temps dans l'exécution de cette fonction. Vu l'importance du temps passé par le processeur dans cette fonction nous avons décidé de la paralléliser. Ainsi le processeur maître qui lui seul possède l'image va la découper et envoyer un morceau de cette image à chaque processeur(scatter). une fois la conversion YUV effectuée chaque processeur échange sa partie convertie de l'image avec les autres(all gather), si bien qu'à la fin tous les processeurs possèdent l'image au format YUV. Il est à noter pour appliquer le filtre de sous-échantillonnage(passage en YUV 4:2:0 ou 4:2:2) nous avons besoin de connaître une zone plus grosse que la zone qu'on veut convertir c'est pourquoi les zones envoyées par le scatter se chevauchent, ce n'est pas le cas pour les zones récupérées par le all gather.

3.1.2 Répartion des données

Chaque processeur se voit affecter un certains nombres de macroblocs répartis aléatoirement sur l'image, ceci afin de mieux équilibrer la charge. Le tableau macroblock_to_process[nb_macroblock] contient la répartition de travail: macroblock_to_process[i] contient le numéro du processeur qui doit traiter la portion d'image correspondant au macrobloc i.

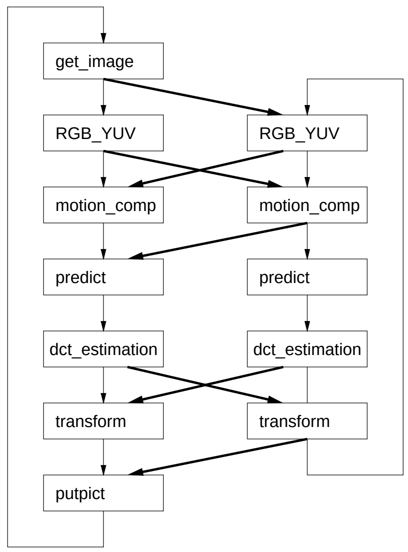


FIG. 3.1 - *Version 1*

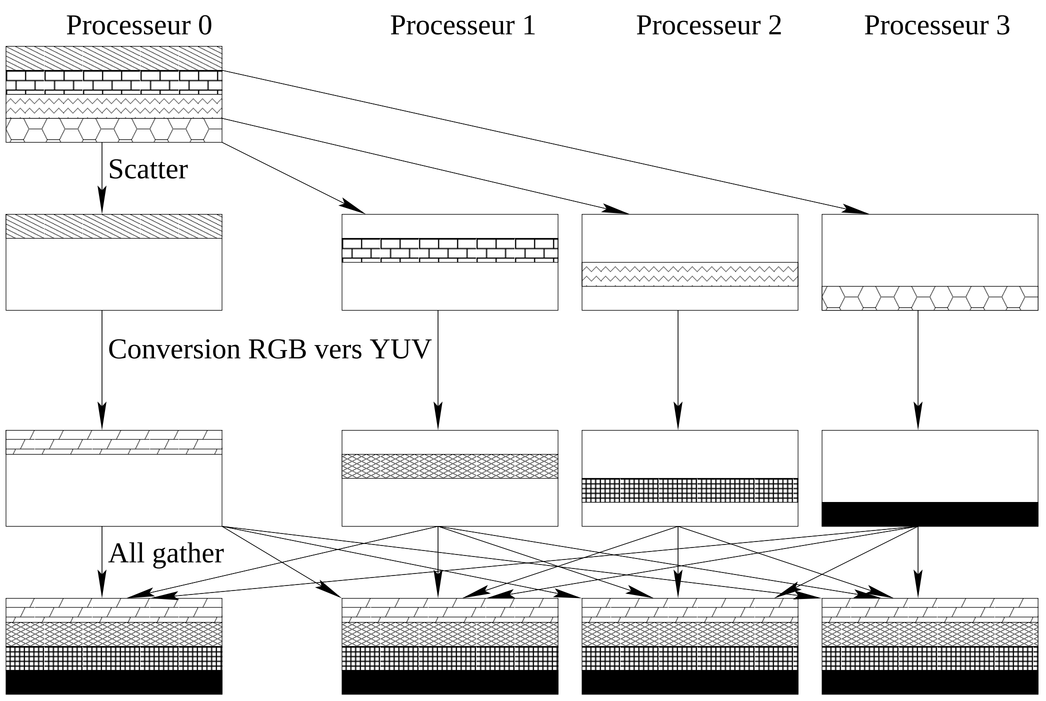


FIG. 3.2 - *La conversion YUV*

3.1.3 Compensation de mouvement

Les processeurs effectuent donc la compensation de mouvement sur leurs zones de travail. Une fois cela fait les processeurs esclaves renvoient les informations sur leurs macroblocs vers le processeur maître (les structures mbi_info). Pour cela la fonction **pm2v_gather_macroblock()** est appelée.

3.1.4 Prédiction d'erreur

La prédiction d'erreur est constituée des appels aux fonctions **predict()**,**dct_type_estimation()** et **transform()**. De même, les processeurs prédisent l'erreur sur leurs zones de travail et s'échangent leurs résultats (les structures blocks) en utilisant fonction **pm2v_gather_blocks()**

3.1.5 Ecriture du flux

L'écriture sur disque et la compression est réalisé par la fonction **putpict()**.
La fonction **putpict()** est intrinsèquement séquentielle puisqu'elle consiste à écrire bit à bit dans le flot de sortie l'image qui vient d'être encodée et à tester toujours la taille du flux encodé qui ne doit pas dépasser une certaine valeur afin de préserver le débit. Or si on dépasse cette taille il faut prendre des mesures qui consistent à dégrader l'image. Gérer un débit constant nécessite une communication régulière ou une perte de précision ce qui serait préjudiciable à la qualité du rendu finale.

3.2 Optimisation de la parallélisation

Malgré la parallélisation de la plupart des fonctions de l'encodeur séquentiel, l'accélération est toujours limitée par **putpict()**. En effet dans la première version de la parallélisation, tous les esclaves doivent attendre la fin de la boucle du maître.

Pour éviter cette latence, nous avons recouvert l'exécution de cette fonction par le calcul de la compensation de mouvement. Pendant que le maître exécute **putpict()** pour l'image courante, les esclaves relancent **motion_estimation()** pour l'image suivante. De plus le maître se verra affecter moins de macroblocs afin de compenser le temps perdu dans lécriture sur disque.

Nous avons dû aussi déplacer la fonction **my_readframe()** chargée de la conversion en YUV et du sous-échantillonnage. En effet cette fonction est parallèle est agit comme une barrière, elle empéchait donc le recouvrement écriture/compensation dans la première version.

Nous avons donc aboutit à un encodeur pratiquement parallèle (fig 3.3).

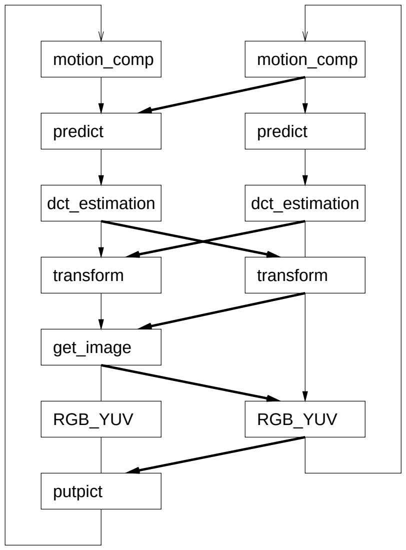


FIG. 3.3 - *Version 2*

Chapitre 4

Régulation dynamique de charge

L'utilisation d'une régulation de charge dynamique permet entre autre de gérer la charge de chaque processeur, ainsi si un processeur est trop chargé, il sera déchargé de la gestion de certains macroblocs. On peut aussi faire tourner l'encodeur sur un réseau de machines hétérogènes puisque chaque machine apportera sa contribution sans ralentir les autres.

Pour mesurer les performances on mesure le temps mis à exécuter la compensation de mouvement(en temps réel et non pas utilisateur puisque l'on veut tenir compte de la charge de la machine) et on le divise par le nombre de macroblocs ayant étés traités.
Nous tenons à jour un enregistrement des 10 dernières performances de chaque processeur afin d'avoir une idée de la charge de ceux-ci. En effet mieux vaut pondérer les derniers résultats puisqu'une erreur dans la mesure de performances coûte cher car elle amène à une distribution fantaisiste de la charge. Des essais sur le SP2 du LABRI, nous ont montré que de mesures de performances sont fortement bruitées d'où l'idée d'un filtre. Nous tenons aussi compte du fait que le maître doit en outre exécuter **putpict()**.
De plus la répartion des macroblocs est totalement aléatoire sur l'image complète pour éviter d'avoir des déséquilibres de charge dans le cas d'une image à moitié noire par exemple puisque les macroblocs d'une moitié de l'image sont alors beaucoup plus vite traités. Si l'on note V_i la vitesse du processeur i en nombre de macroblocs traités par microseconde, N_i le nombre de macroblocs à la charge du processeur i et M le nombre de macroblocs, on notera N' la nouvelle valeur de N pour la prochaine image. Supposons que l'on ait n processeurs.

$$M = \sum_{i=1}^{n} N_i = \sum_{i=1}^{n} N_i'$$

Si Tp est le temps mis pour exécuter putpict par le maître on veut que chaque processeur travaille pendant un temps T pour l'image suivante soit:
pour $i \neq 0$

$$T = \frac{N_i'}{V_i}$$

$$T = \frac{N_0'}{V_0} + Tp$$

D'où:

$$M = \sum_{i=1}^{n} T \times V_i - Tp \times V_0$$

Soit:

$$T = \frac{M - Tp \times V_0}{\sum_{i=1}^{n} V_i}$$

Si $T \leq Tp$ on ne met qu'un seul macrobloc à la charge du maître(au moins un puisqu'il faut conserver des mesures de vitesse afin de pouvoir lui redonner du travail plus tard) et on reprend les calculs. Connaissant ainsi T nous avons N'_i .

$$(\forall i \neq 0)\ N'_i = V_i \times T$$

$$N'_0 = (T - Tp) \times V_0$$

Chapitre 5

L'architecture du projet

Nous avons commencé, comme nous l'avions déjà dit plus haut par éliminer toutes les variables globales du fichier global.h et nous avons crée un fichier .h pour chaque fichiers .c afin d'exporter les fonctions qui le nécessitaient.

Nous avons ensuite crée le fichier pm2v.h qui contient les fonctions de la bibliothèque.Toutes les fonctions que nous avons écrites commencent par "pm2v_" et leur code se trouve le plus souvent dans pm2v.c. Notre bibliothèque doit pouvoir supporter MPI comme PVM. Donc pour faciliter le portage nous avons regroupé les fonctions de communication dans le fichier com.c. Pour le moment le programme ne supporte que MPI mais toutes les fonctions de communications peuvent être réecrites pour PVM. En effet nous n'appelons dans le reste du code que des fonctions que nous avons écrites comme **pm2v_broadcastparam** par exemple. Voici les fonctions communications de notre bibliothèque:

- void pm2v_broadcastparam(pm2v_stream * stream,char * param);
  est exécutée par le maître afin d'informer les esclaves sur les paramètres de compression.
- void pm2v_getparam(pm2v_stream * stream,char ** param);
  est exécutée par les esclaves pour obtenir les paramètres de compression.
- void pm2v_barrier(pm2v_stream * stream);
  sert à créer une synchronisation entre tous les processeurs.
- void pm2v_bcast_load(pm2v_stream * stream);
  le maître envoit à chaque processeur la table **macroblock_to_process** contenant la nouvelle distribution de charge.
- void pm2v_gather_time(pm2v_stream * stream);
  le maître collecte les performances de chacuns des autres processeurs.
- void pm2v_reduce_macroblock(pm2v_stream* stream);
  tous les processeurs récupèrent les macroblocs infos (mbinfo) mis à jour par les autres. Pour des régulations de charge, chaque processeur se voit assigné de façon aléatoire des macroblocs. Les portions d'images assignées à un même processeur sont donc discontigües. Pour pouvoir l'envoyer en une seule fois, les macroblocs sont donc recopiés dans un vecteur de façon contigües, puis envoyer.
- void pm2v_reduce_block(pm2v_stream* stream);
  tous les processeurs récupèrent les blocks mis à jour par les autres. De même ils sont recopiés dans un vecteur de façon contigüe avant envoi.

- void pm2v_scatter_image(pm2v_stream * stream,char * src,int width,int height);
  le maître envoit à chaque processeur un morceau de l'image à convertir en YUV.
- void pm2v_gather_image(pm2v_stream * stream,char * src,int width,int height);
  tous les processeurs récupèrent les morceaux d'image mis à jour par les autres
- void pm2v_init_com(pm2v_stream* stream);
  initialise les communications.

Le passage de paramètre ne se fait plus par fichier mais à l'aide d'une chaîne de caractère que nous analysons à l'aide de lex et yacc. D'où la présence des fichiers params.h params.l params.y params_init.h . Nous avons choisi lex et yacc car ils permettent un traitement plus souple des structures variables puisque les structures mises en jeu varient beaucoup avec les options de compression. Le fichier dynamical_load_balancing.c abrite les fonction liées à l'équilibrage de charge dynamique.

Chapitre 6

Extensibilité

6.1 Charge en calcul

Cas d'une machine parallèle homogène. Soit V la vitesse d'un processeur (en macroblocs par seconde),n le nombre de processeurs et M le nombre de macroblocs à traiter. On cherche le temps minimal T pour l'exécution de putpict() menant à une telle situation . Le maître exécute putpict() et traite un macrobloc alors que chaque esclave traite (M-1)/(n-1) macrobloc. On a un problème de performance si:

$$T + \frac{1}{V} \;\geq\; \frac{M-1}{(n-1) \times V}$$

Ce qui prouve que à processeur donné(V fixé) et format de compression donné(M fixé) on a une valeur maximale de n pour de bonne performance:

$$(n-1) \;\leq\; \frac{(M-1)}{V \times (T + \frac{1}{V})}$$

Or T dépend de la vitesse du processeur maître donc

$$T = \frac{\alpha}{V}$$

α étant homogène à un nombre de macroblocs on peut dire putpict() est équivalent au traitement de α macroblocs. soit :

$$n-1 \;\leq\; \frac{(M-1)}{V \times (\frac{\alpha}{V} + \frac{1}{V})} \;\sim_{M \gg 1}\; \frac{M}{\alpha}$$

La valeur limite de n étant

$$n-1 \;=\; \frac{M}{\alpha}$$

Dans ce cas précis le maître traite α macroblocs (c'est **putpict()**) et chaque esclave traite $\frac{M}{n-1}$ soit α macroblocs, c'est l'équilibre parfait. Donc on a deux cas de figure: soit on n'est pas obligé d'utiliser tous les processeurs et alors on utilise le nombre optimal de processeurs, soit il faut trouver un moyen de paralléliser **putpict()** ce qui semble difficile vu les raisons déjà invoquées plus haut.

6.2 Coût en communication

Nous allons dans cette section décrire toutes les communications utilisées dans notre bibliothèque pour essayer d'évaluer le coût des comunication et leur importance dans l'extensibilité.

Soit M le nombre de macroblocs Mb le nombre de blocs, on a Mb = 6.M en YUV 4:2:0. Un struct mbinfo a une taille de 92 octets. Un bloc a une taille de 128 octets. Soit Mm le nombre de macrobloc affectés au maître et n le nombre de processeurs. Regardons les quantités de données émises et reçues pour chaque fonction de comunication :

– **motion_estimation()** : utilise **pm2v_reduce_macroblock()** ce qui correspond à la réception de (M-Mm).92 octets par le maître .
– **dct_type_estimation()** utilise lui aussi **pm2v_reduce_macroblock()**.
– **transform()** utilise **pm2v_reduce_block()** ce qui correspond à la réception de (M-Mm).6.128 octets par le maître .
– **pm2v_load_balance()** utilise :
  – **pm2v_gather_time()** : réception d'un double(représentant la vitesse) par le maître pour chaque processeurs soit (n-1).8 octets.
  – **pm2v_bcast_load()** : 4 broadcasts soit en tout (2.n+2.M).4 octets.
– **readframe()** utilise :
  – **pm2v_scatter_image()** : le maître envoie à chaque esclave (hauteur.largeur).3/n octets puisque l'on découpe l'image pour la conversion en YUV.
  – **pm2v_gather_image()** on récupère l'image en YUV elle ne fait donc plus que (hauteur*largeur).1,5 octets. chaque processeur envoit le fruit de son travaille ((hauteur.largeur).1,5/n) à tous les autres donc il a circulé en tout (hauteur.largeur).1,5.n octets.

Soit pour le calcul d'une image:

$$352 \times (M - M_m) + (n-1) \times 8 + (n-1)(2 \times n + 2 \times M) \times 4 + taille \times (3 + 1,5 \times n)\ octets$$

Dans le cas particulier de nos mesure de test:
M=920, n=4, M_m=160, hauteur=720,largeur=352
Soit un débit de 3217176 octets (3Mo) pour le calcul d'une image, avec 2280960 octets(2.17 Mo) pour **readframe()** et 936216 octets(914Ko) pour le reste.

Un processeur donné reçoit (ou émet) :

$$2 \times (M - M_m) \times 92/(n-1) + (M - M_m) \times 768/(n-1) + 8 + (2 \times n + 2 \times M) \times 4 + taille \times 3/n + taille \times 1,5 \times 2.((n-1)/n) = 1198973\ octets\ soit\ 1,14\ Mo$$

pour le cas limite où M_m=0 on a:

$$com = 767872 + 8 \times n + \frac{706744}{n-1}\ octets\ images$$

D'aprés la formule nous voyons que le cout en communication évolue au mieux linéairement avec le nombre de processeurs pour le calcul d'une image. De plus quand le nombre de processeurs augmentent, le temps de calcul évolue en 1/n. Nous avons donc un coup de communication égal à:

$$C_{dbit} \sim n^2$$

Pour espérer du temps réel il faut reçevoir (et émettre) ces données en moins de $\frac{1}{25}$ ème de seconde soit des débits de:
35 Mo/s pour 2 processeurs
23 Mo/s pour 4 processeurs

20 Mo/s pour 8 processeurs
19 Mo/s pour 16 processeurs

On peux constater que ces débits sont irréalistes sur la plupart des réseaux courants, il est donc nécessaire d'avoir un réseau dédié. Il est à noter qu'en dimminiuant la taille de l'image par 4 on divise presque le débit minimum par 4, on utilise pour le moment la taille des images télévision.

Chapitre 7

Utilisation de la bibliothèque pm2v

7.1 installation

La bibliothèque **pm2v** nécessite préalablement les logiciels suivants.

- Flex Version 2.5.4.
- Bison Version 1.25.
- Un compilateur C ANSI.
- Une bibliothèque implémentant la norme MPI.

L'installation de la bibliothèque se fait trés simplement. Il suffit d'exécuter les commandes makedepend et make. Le nom de la bibliothèque est **libpm2v.a** et le nom du fichier d'include **pm2v.h**

Le programme à été testé avec succés sur les plateformes suivantes:
AIX, Solaris, GNU Linux.

7.2 Interface

La bibliothèque pm2v basé sur le travail du **MPEG simulation group** possède une API très sommaire.

7.3 pm2v_open()

Prototype:
int pm2v_open(pm2v_stream * stream, char * param, FILE * output
, pm2v_communicator com, int master_id, int my_id, int nb_proc);

Cette fonction doit être appelé avant tout appel à pm2v, en effet elle est chargée d'initialiser le compresseur.

Paramètres:
stream: pointeur sur une structure de type pm2v_stream allouée par l'utilisateur. Le contenu de cette structure est réservé à l'usage de la bibliothèque.
param: Pointeur sur une chaîne de caractère décrivant les paramètres de compression.
output: Flux dans lequel sera envoyé les données MPEG.
com: Structure de donnée décrivant la machine parallèle (un MPI_Communicator dans le cas d'une compilation pour MPI).

master_id: Le numéro d'identification du maître.
my_id: Le numéro d'identification du processus courant.
nb_procs: Le nombre total de processeurs effectuant la compression.

7.4 pm2v_add_image()

Prototype:
int pm2v_add_image(pm2v_stream * stream, unsigned char * img)
Cette fonction encode une image dans le flux courant. L'image est mise dans la queue d'encodage avant d'être finalement compressée.
Paramêtres:
stream: Pointeur sur une structure de type pm2v_stream remplie préalablement par l'appel à **pm2v_init()**.
img: Pointeur sur l'image à compresser. L'image doit se trouver au format RGB non compressé, sous la forme d'une succession de triplets RGB. Chaque composante RGB est codée sur 8 bits, et l'image est linéairement disposée par lignes.

7.5 pm2v_close()

Prototype:
int pm2v_close(pm2v_stream * stream)
Cette fonction encode les dernières trames qui restent éventuellement dans la queue et ferme le flux MPEG.
Paramêtres:
stream: Pointeur sur une structure de type pm2v_stream rempli préalablement par l'appel à **pm2v_init()**.

Chapitre 8

Les résultats

Nous avons effectués tous nos tests avec une série de 94 images de résolution 720x352 représentant environ 10 secondes de vidéos.
De plus, nous avons constaté que la régulation de charge met au moins 20 à 30 images à se stabiliser.

8.1 IBM SP2

Nous avons effectué la plupart de nos tests sur le SP2 du LaBRI, qui est constitué de 16 nœuds.

Nombre de processeurs	Temps (secondes)	Efficacité (%)	Cout (secondes)
1	528	100	0
2	275	96	22
3	198	89	66
4	161	82	116
5	140	75	172
6	123	72	210
7	115	66	277
8	108	61	336
9	105	56	417
10	101	52	482
11	100	48	572
12	100	44	672
13	99	41	759
14	99	38	858
15	98	36	942
16	98	34	1040

Sur la graphique 8.1, nous voyons que notre projet est extensible jusqu'a environ 8 processeurs, au delà le processeur maître n'effectue que lécriture sur disque. Ces résultats sont confirmées par le graphique 8.2 représentant l'efficacité.

L'efficacité théorique tracé sur la figure 8.2 et en fait l'efficacité dans le cas de la saturation.

8.2 Sun enterprise 450

Les tests ont portés sur une machine de l'ENSERB, comportant quatres processeurs UltraSparc SMP à 400 Mhz.

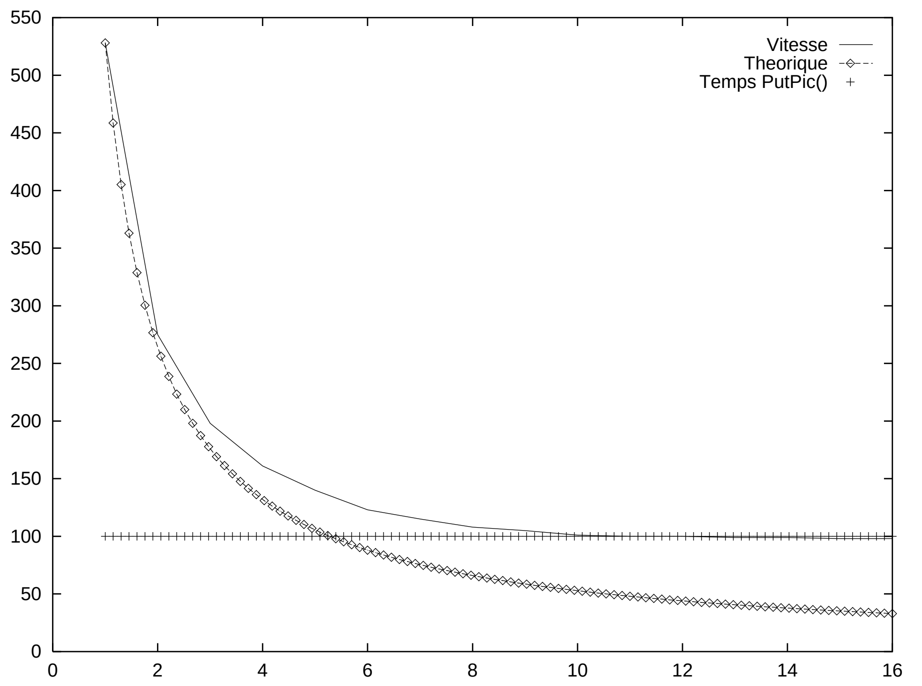


FIG. 8.1 - *Vitesse en fonction du nombre de processeurs.*

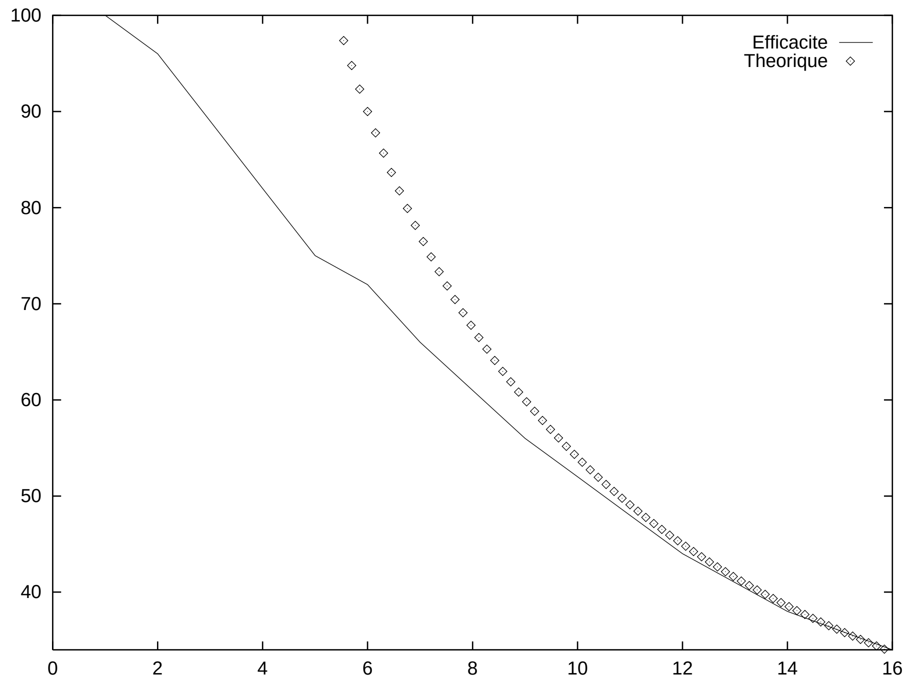


FIG. 8.2 - *Efficacité en fonction du nombre de processeurs.*

Nous avons utilisé une version de MPI disponible librement sur internet **MPICH (Argonne Laboratory)**
Nous avons mesuré les performances suivantes:

Nombre de processeurs	Temps réel (secondes)	Temps cpu (secondes)	efficacité (%)(
1	156	156	100%
2	98	75	80%
3	75	50	70%
4	60	37	65%

Nous remarquons que les performances baissent beaucoup plus rapidement que sur le sp2, cela est d'autant plus suprenant que l'architecture SMP devrait réduire énormément le cout des communications. Le fait que ce serveur ne soit pas une machine dédié au calcul doit beaucoup jouer dans ces résultats. En effet les performances varient beaucoup en fonction de la charge, même si celle ci est trés faible. La politique de l'ordonnanceur ne doit pas favoriser notre programme.

8.3 Réseau hétérogène

Nous avons utilisé deux serveurs SUN avec de processeurs UltraSparcs reliés tous deux par un réseau Ethernet à 100 Mbit/s. L'un des serveur est équippé avec 4 processeurs à 400 Mhz, l'autre avec 2 processeur à 200 Mhz.

Processeurs sur M1	Processeur sur M2	Temps réel
1	0	156
0	1	315
1	1	134
2	1	95
1	2	92
2	2	86

On peux constater que les performances sont équivalentes à l'utilisation du SMP.

8.4 Réseau de stations de travails

Nous avons utilisé des stations UltraSparc II, reliés par un réseau ethernet 10 Mbit/s.

Nombre de stations	Temps réel	efficacité
1	187	100%
2	127	73%
3	160	40%

Nous pouvons constater que le coût de la communication est trés important sur ce type de réseau. Le coût de ce type de réseau est modique, en effet c'est le type d'installation le plus fréquemment utilisé. De ce fait bien que l'accélération soit trés faible, la possiblilté d'utiliser ce type de réseau est intéréssante.

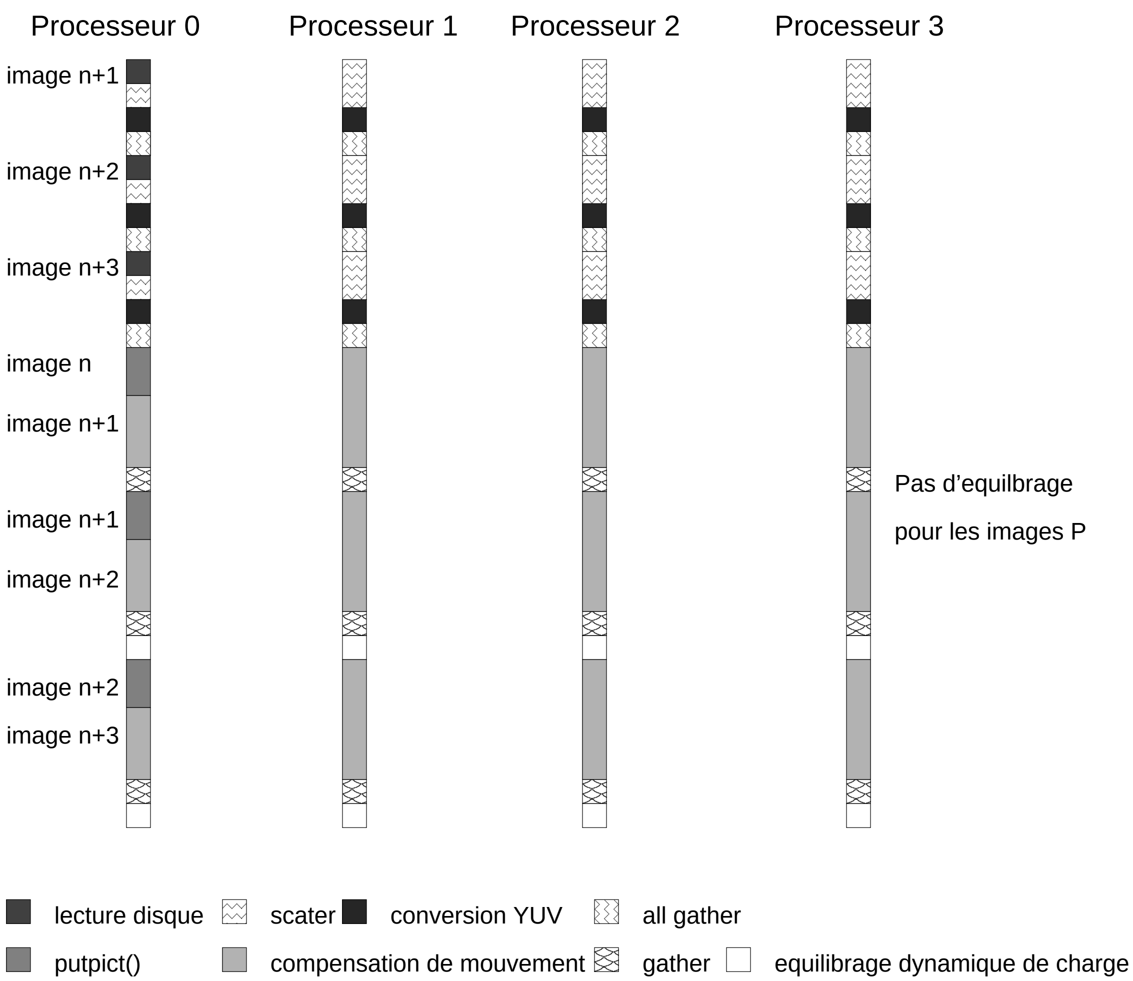


FIG. 8.3 - *Diagramme temporel*

Chapitre 9

Améliorations possible

On le voit dans les tests de performance il y a deux facteurs limitants:

- la séquentialité de putpict()
- le débit du réseau inter-processeurs

9.1 putpict()

Dans notre bibliothèque putpict() est séquentielle et s'éxècute toujours sur le processeur maître. Ainsi on peut arriver dans le cas extrême ou même déchargé au maximum(1 macrobloc seulement) le processeur maître met plus de temps à exécuter putpict() que les autres processeurs à traiter leur macroblocs. Dans ce cas à la prochaine synchronisation tous les processeurs vont attendre le maître ce qui crée une baisse de performance (figure 9.1).

9.2 le réseau

Pour le traitement de chaque image nous distribuons l'image à traiter sur tous les processeurs ce qui n'est pas sans handicaper les machines dont le réseau d'interconnexion est lent. Nous avons choisi de distribuer les macroblocs de manière aléatoire sur les processeurs afin d'éviter les déséquilibre de charge, mais nous aurions pu aussi donner à chaque processeur une portion connexe des macroblocs . Ainsi en ne donnant à chaque processeur que la partie de l'image recouvrant la zone de macroblocs concernée(avec une marge car on a une valeur maximale pour la taille des vecteur de mouvement) on limite les communications.

De plus on pourrait recouvrir les communications pour l'envoi d'image par le calcul. En effet les processus pourraient calculer pendant la reception de l'image suivante. Cela pourrait se faire par des communications persistantes, mais dans ce cas on perdrait l'optimisation du au broadcast. Dans la version de MPI II, il est prévu de pouvoir faire des communications collectives asynchrones. En attendant cela peux-être implémenté à l'aide de threads.

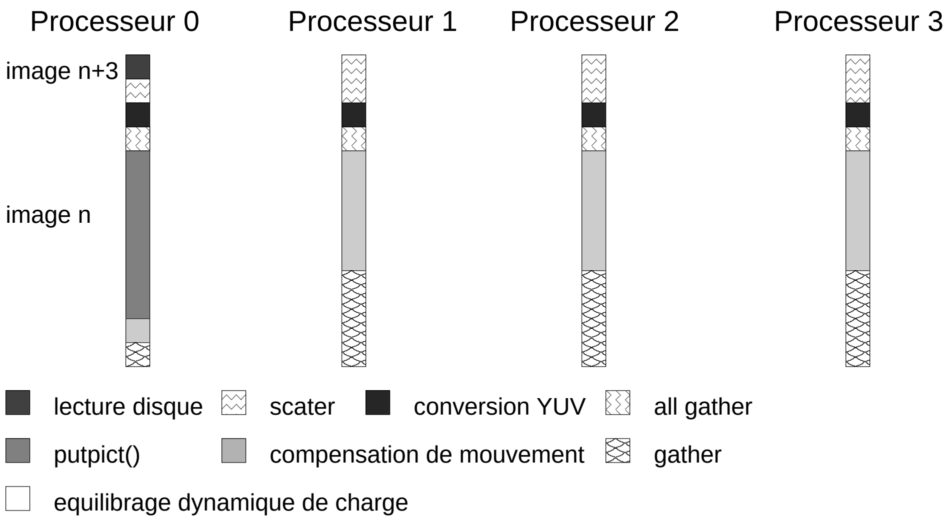


FIG. 9.1 - *Cas de perte de performance dûes à putpict()*

Conclusion

L'encodeur développé lors de ce projet est efficace pour un nombre restreint de processeurs (environ 8) et sur un réseau rapide (au moins 100 Mbit/s). De plus on peux constater que dans le cas de réseau de stations de travail, l'encodeur accélére le calcul dans le cas ou uniquement 2 ou 3 machines effectuent le travail.

Cependant, le projet était difficile puisqu'il demandait des connaissances en traitement de l'image que nous n'avions pas. Il pourrait être judicieux de reprendre le projet avec une équipe consistuée d'un paralléliste et d'un élève de l'option TIC pour tenter d'optimiser encore le code, en particulier pour la paralélisation de **putpict()**.

Ce projet nous aura permis de consolider nos connaissances en conception parallèle. Nos seul regrets portent sur le code existant qui n'était ni efficace ni commenté. Le temps perdu à nettoyer le code aurait pu être mis à profit pour la recherche d'une solution au problème posé par **putpict()**. De plus il aurait été préférable de paralléliser un code plus performant. Nous n'avons pas eu le temps de faire une version utilisant PVM mais notre architecture est conçue pour faciliter le passage de MPI à PVM.

Glossaire

- macrobloc : tableau représentant un carré dans l'image à encoder de 16 par 16 pixels.L'image est codée en YUV.
- compensation de mouvement : utilisation de vecteur de mouvement pour situer un macrobloc dans une image précédente et/ou suivante.
- GOP : group of picture, séquence d'image entre deux images I successives.
- DCT : discret cosine transform, variante de la transformée de Fourrier dicrète.
- iDCT : inverse discret cosine transform. Inverse de la DCT.
- MPEG I : format video défini par la norme ISO/IEC 11172-2.
- MPEG II : format video défini par la norme ISO/IEC 1318-2.
- RGB : format de couleur où chaque pixel est donnée par trois composantes rouge , verte et bleue.
- YUV : format de couleur où chaque pixel est donnée par trois composantes, une de luminance et deux de chrominance.
- MPI : message passing interface, bibliothèque de comunication parallèle.
- PVM : parallel virtual machine, bibliothèque de comunication parallèle.

Bibliographie

[1] *ISO/IEC JTC1/SC29/WG11 N0702 Rev, "Information Technology - Generic Coding of Moving Pictures and Associated Audio, Recommendation H.262", tech. rep., Draft Internation al Standard, Paris*, 1994.

[2] *ISO/IEC 11172-2, "Information Technology - Coding of Moving Pictures and As sociated Audio for Digital Storage Media at up to about 1,5 Mbit/s - Video", tech. rep., Gene va*, 1993.

[3] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "Pvm and mpi: a comparison of features," tech. rep., May 1996.

[4] *MPI: A Message Passing Interface Standard*, 1995.

[5] L. Lamport, *latex a document preparation system*.