Universitat de Lleida
Escola Politècnica Superior
Enginyeria Tècnica en Informàtica de Gestió
Treball de final de carrera

Implementació i avaluació del paral·lelisme de dades a l'aplicació BASIZ

Autor: Eloi Gabaldon Ponsa
Directors: Fernando Guirado
Concepció Roig
Setembre de 2009

Índex

1- Introducció 4
1.1 Objectius 5
2- Descripció de l'aplicació BASIZ 6
2.1 Funcionament 6
2.2 Tecnologies que utilitza 12
2.2.1 Mpeg2dec 0.4.1 12
2.2.2 CImg 1.3.1 13
3-Augment de paral·lelisme de BASIZ 14
3.1 Tasques a realitzar 14
3.2 Implementació del paral·lelisme de dades 16
3.2.1 Programa en Sèrie 16
3.2.2 Programa en paral·lel 18
3.2.3 Generador de XML per calcular els mappings 29
4- Preparació de l'entorn 32
4.1 Instal·lació 32
4.2 Compilació i execució 32
5- Experimentació 35
5.1 Proves de rendiment 35
5.2 Proves de verificació de resultats 37
6- Conclusions 39
7- Bibliografia 40

Índex de figures

Figura 1: Imatge original i sortida generada per BASIZ.....6
Figura 2: Estructura de tasques de l'aplicació BASIZ.....7
Figura 3: Exemple d'imatge d'entrada a BASIZ.....8
Figura 4: Imatges resultants després de fer la selecció del color.....8
Figura 5: Difuminat de nivell 3 9
Figura 6: Difuminat de nivell 4 9
Figura 7: Difuminat de nivell 5 10
Figura 8: Resultat de la suma dels difuminats per a cada color 10
Figura 9: Imatge obtinguda després de la suma de difuminats 11
Figura 10: Imatge codificada en HSI 11
Figura 11: Imatge obtinguda després de la comparació amb els valors de llindar 12
Figura 12: Imatge resultant després d'aplicar BASIZ 12
Figura 13: Exemple d'imatge tallada per realitzar els difuminats de nivell 3, 4 i 5 respectivament en 2, 4 i 6 trossos. 19
Figura 14: Imatges que no es poden sumar 19
Figura 15: Imatges que si que es poden sumar.....20
Figura 16: Imatge després de sumar els diferents colors 20
Figura 17: Imatge resultat del nou algoritme 21
Figura 18:Imatge on es mostra l'esquema per una determinada configuració 23
Figura 19: Gràfic resultant de l'arxiu .dot anterior 31
Figura 20: Gràfica de temps d'execució del BASIZ original 35
Figura 21: Gràfic amb el còmput de les tasques per una configuració 2-2-2.....36
Figura 22: Gràfic on s'observa la millora de rendiment per diferents configuracions....36
Figura 23: Configuració amb resultats semblants a l'original 37
Figura 24: Imatge amb resultats diferents de l'original.....37

1- Introducció

En l'àmbit de la informàtica sempre s'ha intentat millorar el rendiment d'aquelles aplicacions que requereixen un temps de còmput molt elevat. A mesura que la informàtica va anant avançant es va desenvolupar la forma de computació anomenada paral·lelisme [1].

Aquesta forma de computació es basa en la partició de problemes complexos en diferents problemes més senzills. Així aquests es poden resoldre simultàniament en diferents processadors o màquines tot obtenint el mateix resultat en menys temps.

Des de que va aparèixer el paral·lelisme, s'han creat diversos paradigmes per tal de poder aplicar-lo. El paradigma de programació per pas de missatges és aquell que permet a aplicacions distribuïdes executar-se en màquines diferents.

Amb aquest paradigma es permet la separació en varies tasques de forma que cada tasca realitzi alguna part significativa de l'algoritme de l'aplicació. Aquestes tasques, un cop han acabat de processar la seva part, envien el resultat a la següent tasca i d'aquesta manera l'algoritme es va realitzant sense importar si les tasques s'han realitzat en la mateixa màquina o en màquines diferents.

La llibreria de programació MPI (Message-Passing Interface) [2], implementa el pas de missatges de manera estandarditzada. Utilitzant aquest sistema, al programador no li cal preocupar-se de l'arquitectura de les diferents màquines on s'executarà el programa ni de com distribuir el còmput en les màquines disponibles.

Des de la seva aparició, moltes aplicacions han estat desenvolupades utilitzant aquesta llibreria, una d'aquestes aplicacions es BASIZ (Bright and Saturated Image Zones) [3]. Aquesta aplicació, que va ser creada l'any 2001 dins del grup d'Arquitectures Paral·leles de la Universitat Autònoma de Barcelona. És una aplicació paral·lela que s'encarrega de tractar imatges mitjançant un algoritme que detecta les zones més brillants o amb més intensitat de color. Aquesta aplicació estava desenvolupada originalment en el llenguatge C i utilitzava la tecnologia PVM (Paral·lel Virtual Machine) [4], un sistema de pas de missatges que es va crear l'any 1989 a la universitat de Tennessee.

L'any 2008 Jordi Píriz, estudiant d'Enginyeria Tècnica en Informàtica de Sistemes a la Universitat de Lleida, va adaptar l'algoritme original del BASIZ per tal que en lloc de utilitzar la tecnologia PVM utilitzés MPI, ja que aquesta tecnologia ha esdevingut, amb el temps, el estàndard de pas de missatges [5].

A més d'implementar l'algoritme de BASIZ utilitzant MPI, Jordi Píriz va millorar considerablement el rendiment de l'aplicació, ja que va utilitzar tecnologies més noves com ara la llibreria Cimg en lloc de utilitzar la URT pel tractament de imatges. Aquesta llibreria aporta un gran repertori de funcions que aconsegueixen el mateix resultat en un temps més baix ja que al ser més nova el tractament de imatges digitals ha avançat i millorat.

Una de les tècniques més utilitzades per augmentar el grau de paral·lelisme d'una aplicació consisteix en aplicar el paradigma de programació anomenat paral·lelisme de

dades. Aquest paradigma consisteix en separar el volum de dades a tractar en blocs i aplicar en cada bloc el mateix algoritme.

Per aplicar aquest paradigma es necessita que l'aplicació permeti separar les dades en blocs independents. Això en l'aplicació BASIZ es pot realitzar tot separant les imatges en bocins més petits i executant l'algoritme per cadascun d'aquests bocins com si es tractés de la imatge sencera. Després només cal tornar a ajuntar les imatges petites formant una altra vegada la imatge original. A l'hora de fer aquest procés caldrà comprovar que el resultat final no quedi modificat.

1.1 Objectius

Aquest projecte té com a objectiu principal augmentar el rendiment de l'aplicació BASIZ mitjançant l'aplicació del paral·lelisme de dades en les tasques on sigui viable.

Per això, es realitza una anàlisi de quines són les tasques que permeten l'aplicació d'aquest paral·lelisme i quines no. Es verifica també que allí on s'aplica no fa canviar el resultat final.

Es fa la implementació del paral·lelisme a l'aplicació BASIZ i es fa una avaluació de rendiment.

Per dur a terme els objectius marcats aquesta memòria està organitzada en els següents capítols:

-Capítol 2: Es descriu el funcionament de l'aplicació BASIZ.
-Capítol 3: Es descriu el treball fet per augmentar el paral·lelisme de BASIZ.
-Capítol 4: S'explica com preparar l'entorn per executar BASIZ.
-Capítol 5: Les proves realitzades i els resultats.
-Capítol 6: Conclusions del treball.

2- Descripció de l'aplicació BASIZ

BASIZ és una aplicació de tractament d'imatges digitals que detecta les zones importants de la imatge, és a dir aquelles que són més visibles a causa del seu nivell de lluminositat o intensitat de color. Així la sortida de l'aplicació seria com es pot observar en la Figura 1, a dalt la imatge original i a baix les zones enquadrades són les més importants de la imatge.

Figura 1: Imatge original i sortida generada per BASIZ

2.1 Funcionament

Per localitzar les zones amb més lluminositat i color, BASIZ utilitza un algoritme que realitza els següents passos:

Pas 1. Descodifica el fitxer mpg i per a cada imatge obtinguda:
Pas 2. Separa la imatge en els 3 canals de color RGB (Red, Green, Blue).
Pas 3. Per a cada imatge de cada canal, s'apliquen els difuminats de nivell 3,4 i 5.
Pas 4. Es sumen els 3 difuminats de cada color
Pas 5. Es sumen les imatges resultants de cada color
Pas 6. Es converteix la imatge de format RGB a HSB (Hue ,Saturation, Brightmess).
Pas 7. Es detecten els llindars de lluminositat i color.
Pas 8. Amb la imatge anterior, es marquen les zones més importants sobre la imatge original i es mostra per pantalla.

Aquest algoritme es du a terme amb un conjunt de 20 tasques (T0,...,T19) que s'organitzen com es mostra en la Figura 2.

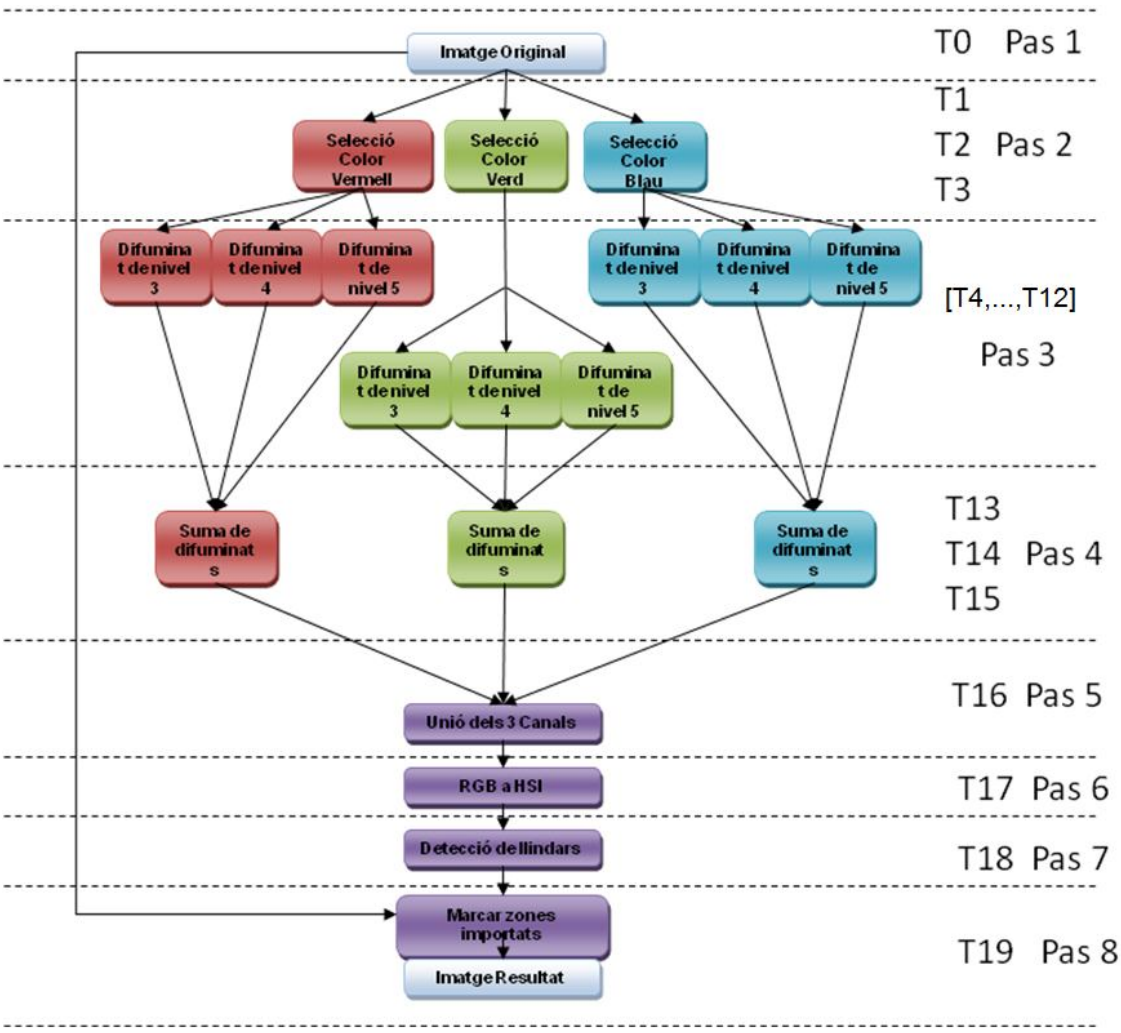

Figura 2: Estructura de tasques de l'aplicació BASIZ

A continuació s'explica detalladament el funcionament de cadascun del passos de l'aplicació i les tasques que el formen.

Pas 1. Extracció del fitxer en MPG (T0).

> Aquest pas conté una única tasca, T0, que s'encarrega de descodificar els fitxers de vídeo en mpg i obté les imatges que posteriorment seran tractades. A la Figura 3 es mostra una possible imatge d'entrada que servirà d'exemple per a l'explicació.

Figura 3: Exemple d'imatge d'entrada a BASIZ

Pas 2. Selecció del canal de color (T1, T2, T3).

Aquestes tasques extreuen els diferents canals de color de la imatge original: vermell, verd i blau. Un cop fet s'obtenen tres imatges. Al fer això el volum de dades a tractar es redueix. Això indica que l'aplicació ja disposa d'un petit nivell de paral·lelisme de dades. En la Figura 4 es pot observar les imatges resultants en la sortida d'aquestes tasques.

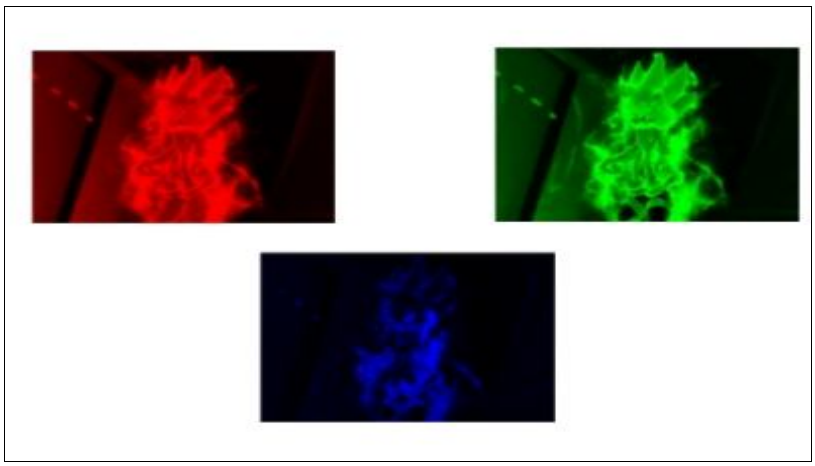

Figura 4: Imatges resultants després de fer la selecció del color.

Pas 3. Difuminats a varis nivells (T4 ,..., T12).

Les tasques d'aquest pas realitzen un difuminat de diferent nivell sobre les imatges resultants de les tasques anteriors.
Així doncs les tasques T4, T7 i T10 realitzen el difuminat de nivell 3 (Figura 5), les tasques T5, T8 i T11 difuminen amb nivell 4 (Figura 6) i les tasques T6, T9 i T12 ho fan amb nivell 5 (Figura 7).
Aquests difuminats es realitzen mitjançant un filtre Canny – Deriche. Aquest filtre s'utilitza fàcilment gràcies a la incorporació de la llibreria Cimg al projecte. A més aporta una gran millora de rendiment respecte la aplicació de BASIZ amb PVM, ja que aquest filtre es menys costos que utilitzar la multiplicació de matrius, que era el mètode utilitzat en el BASIZ original.

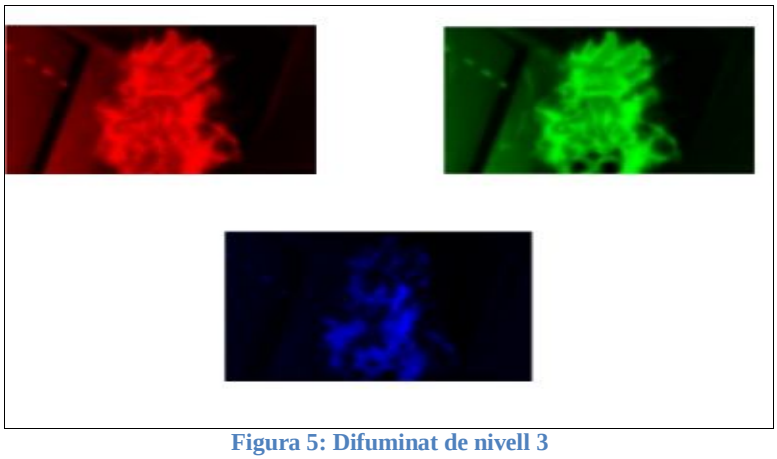

Figura 5: Difuminat de nivell 3

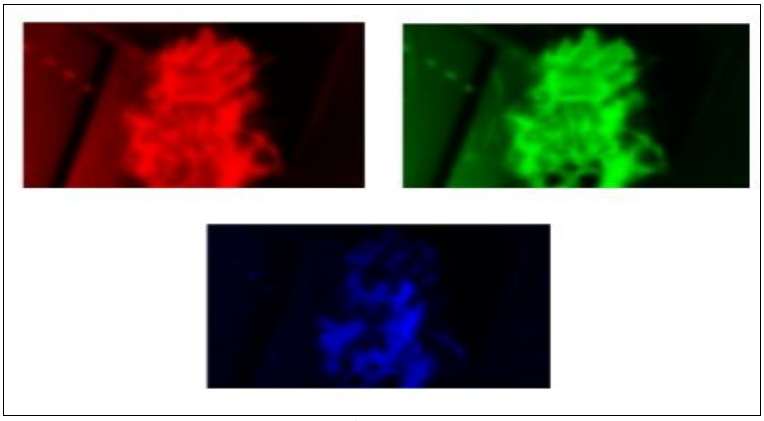

Figura 6: Difuminat de nivell 4

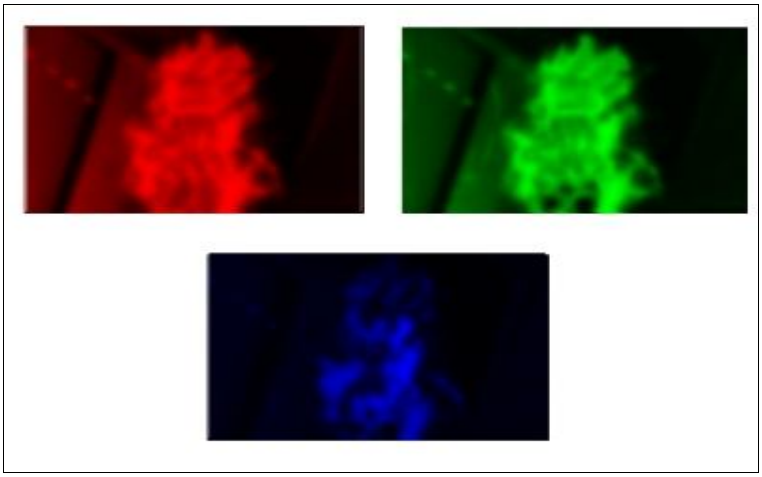

Figura 7: Difuminat de nivell 5

Pas 4. Suma dels difuminats (T13, T14, T15)

En aquest pas es sumen les tres imatges resultants dels difuminats de cada canal. En la Figura 8 es pot observar el resultat.

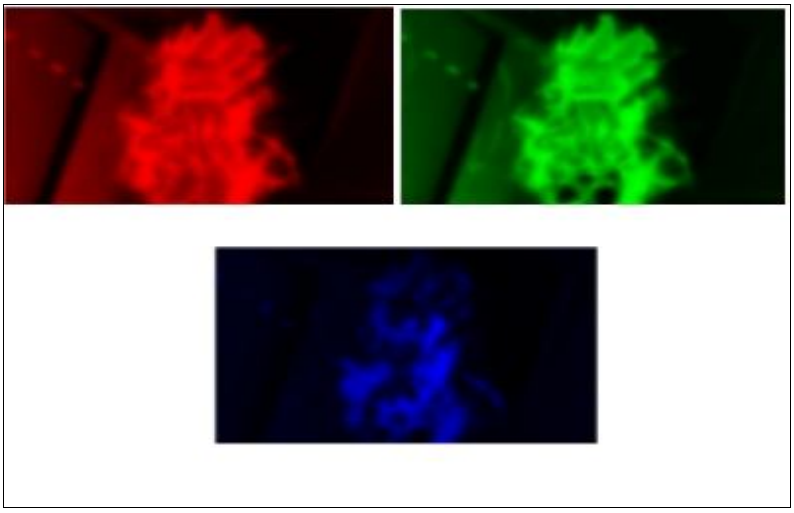

Figura 8: Resultat de la suma dels difuminats per a cada color

Pas 5. Suma dels canals de color (T16)

Aquest pas suma els tres canals de color obtenint com a resultat una imatge en color i difuminada. En la Figura 9 es pot observar que les zones més importants continuen apareixent clarament mentre que les zones poc importants es confonen amb el fons de la imatge.

Figura 9: Imatge obtinguda després de la suma de difuminats

Pas 6. Conversió de RGB a HSB (T17)

Aquesta tasca converteix la imatge anterior de format RGB a format HSB (Figura 10). El format RGB no es pot utilitzar per la detecció de zones importants (els valors que guarda són destinats a indicar la quantitat de color vermell, groc i blau existents en cada píxel). En canvi, això no passa si utilitzem el format HSB ja que aquest guarda valors destinats al nivell de intensitat de color i lluminositat.

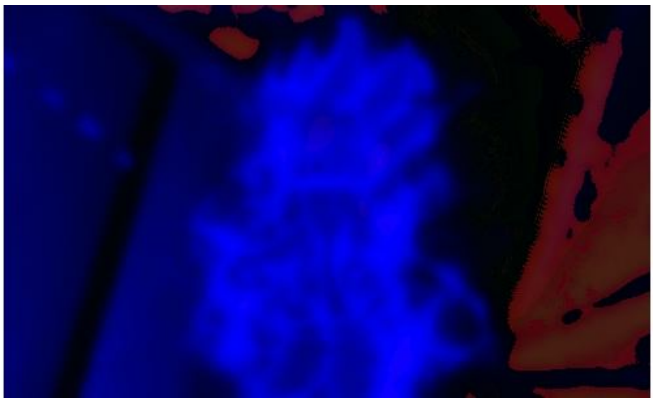

Figura 10: Imatge codificada en HSI

Pas 7. Detecció de llindars (T18)

Un cop la imatge està en format HSB es poden calcular els llindars. Es sumen tots els valors de la imatge i es fa la mitja dels valors de saturació i intensitat. Aquests valors esdevenen el llindar per decidir si un píxel es significatiu o no. Un cop fet, es tornen a comprovar tots els píxels, marcant els que superen el llindar i deixant a negre els que no (Figura 11).

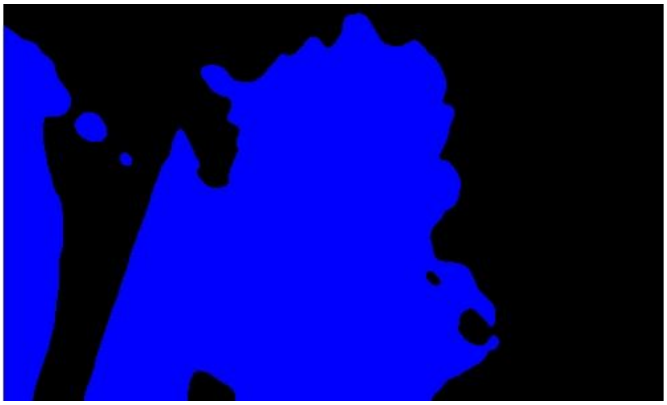

Figura 11: Imatge obtinguda després de la comparació amb els valors de llindar

Pas 8. Marcar les zones i mostrar per pantalla (T19)

Aquesta tasca marca amb requadres, a la imatge original, les zones importants que superen un determinat tamany de píxels marcades en la imatge anterior com es pot observar en la Figura 12.

Figura 12: Imatge resultant després d'aplicar BASIZ

2.2 Tecnologies que utilitza

En la implementació que va desenvolupar Jordi Píriz, es van introduir aquestes dues llibreries: CImg i libmpeg2, que explicaré a continuació.

2.2.1 Mpeg2dec 0.4.1

La llibreria libmpeg2 es la llibreria utilitzada per descodificar el fitxer en mpg i obtenir les imatges. Aquesta llibreria va ser utilitzada també en el BASIZ original i en la meva modificació he mantingut la mateixa llibreria i el mateix sistema d'extracció d'imatges.

2.2.2 CImg 1.3.1

La llibreria CImg va ser introduïda en la versió MPI de BASIZ. En la meva modificació he mantingut també aquesta llibreria, tot i que he actualitzat la versió ja que la versió anterior donava alguns problemes. Aquesta llibreria conté moltes funcions de tractament d'imatges molt útils i que han simplificat molt el procés d'elaboració del projecte.

3-Augment de paral·lelisme de BASIZ

En aplicacions de tractament d'imatges, és important intentar reduir el temps de còmput per tal d'aconseguir un flux d'imatges prou ràpid per poder veure els vídeos en temps real.

Una manera d'augmentar el rendiment d'aquestes aplicacions consisteix en aplicar paral·lelisme de dades en les imatges. Però algunes vegades, l'aplicació d'aquest paradigma pot causar deformacions a les imatges tractades.

En aquest projecte ens vam proposar augmentar el grau de paral·lelisme del que disposa l'aplicació BASIZ per tal de estudiar si les diferències entre les solucions eren acceptables. Si els resultats són favorables, es podrà realitzar el paral·lelisme i millorar notablement el seu rendiment.

Des d'un principi vam decidir que les tasques que s'havien de distribuir serien les tasques de difuminat, ja que son les que poden arribar a modificar les imatges de tal manera que el resultat sigui diferent.

Aquest treball s'ha realitzat modificant el projecte de BASIZ amb MPI i per tant els factors com son el llenguatge de programació a utilitzar, l'estructura del programa, i les Tecnologies a utilitzar ja venien marcades pel projecte des d'un bon principi.

Així doncs el llenguatge que utilitzarem per fer el projecte es C i C++, la llibreria utilitzada pel tractament de imatges es la Cimg i per a l'extracció de vídeo s'ha utilitzat libmpeg2.

Tot i que les tecnologies son les mateixes, en el cas de la llibreria Cimg vam haver de actualitzar-la a la versió 1.3.1 ja que la versió 1.2.7 ens donava problemes al finalitzar els processos que obrien una finestra per mostrar imatges.

3.1 Tasques a realitzar

En un principi, quan vaig començar a treballar amb el projecte, vam establir unes tasques ha realitzar per assolir els objectius del projecte, aquestes tasques consistien en:

1 programa en sèrie

Per tal d'acostumar-se a la utilització de les tecnologies utilitzades en la aplicació de BASIZ amb MPI, en l'estructura de l'algoritme de BASIZ i en definitiva en tot el referent al projecte, es va demanar la modificació del programa en sèrie per a que realitzes l'algoritme de BASIZ tot separant les imatges com hauria de fer la meva aplicació paral·lela.

Un cop finalitzat el programa en sèrie es pot dir que ja tenia els coneixements i les eines necessaris per tal de poder afrontar un programa més complex com el BASIZ en paral·lel.

2 programa en paral·lel

Aquesta tasca, a l'igual que el programa en sèrie ha de seguir la mateixa estructura que la versió original de BASIZ tot afegint el necessari per tal d'aplicar a l'aplicació paral·lelisme de dades. De manera que les tasques de difuminat s'executin paral·lelament.

En observar l'estructura de BASIZ amb MPI vam veure que la funció main és un swich on cada procés te un número i executa la tasca corresponent a aquest numero, això era un problema ja que cada vegada que es volia introduir una tasca nova s'hauria d'escriure un codi nou.

Aquest problema el vam solucionar utilitzant un fitxer on s'indica la configuració de les diferents tasques.

Per tal de facilitar la creació d'aquest fitxer i evitar errors es va implementar un petit programa que genera el fitxer en resposta al nombre de processos que es desitgi per cada tipus diferent de difuminat.

3 XML per detectar el mapping més adequat

Un cop el programa en paral·lel funcionés correctament vam veure que hauríem de decidir quin seria el mapping (assignació de tasques a processadors) més adequat per aprofitar el màxim la potència de les màquines.

Per poder fer aquesta comprovació es va demanar realitzar un programa que, obtenint dades del funcionament de l'aplicació en paral·lel, fos capas d'escriure un fitxer XML per tal de poder utilitzar un programa de càlcul de mappings.

Aquest programa a més proporciona un fitxer .dot per tal de poder generar una imatge-gràfic a on es mostri l'estructura en paral·lel de l'aplicació amb la configuració actual.

4 Proves i experimentació

Un cop estigui tot fet, executarem el programa en paral·lel utilitzant diferents configuracions del programa i compararem els resultats obtinguts amb l'execució del programa en paral·lel original. També observarem el temps d'execució de les diferents tasques.

3.2 Implementació del paral·lelisme de dades.

3.2.1 Programa en Sèrie.

En un principi, vam començar amb la realització d'un programa en sèrie, ja que la realització d'aquest es més senzilla i serveix per habituar-se a treballar en el problema en paral·lel.

El desenvolupament d'aquesta aplicació en sèrie es pot dividir en 3 parts que tot seguit son explicades.

3.2.1.1 Creació de la funció per tallar imatges

Per tal de poder tallar les imatges en trossos més petits, que és una tasca necessària per poder separar les tasques de difuminat, vaig buscar en la documentació de la llibreria Cimg per observar si existia alguna funció que servis per aquest fi.

Vaig trobar la funció següent:

```
CImg< T > get_crop (const int x0, const int y0, const int x1, const int y1, const bool border_condition=false) const
```

Com passa amb la majoria de funcions de la llibreria Cimg, els paràmetres de les funcions tenen valors per defecte, de manera que en algunes ocasions no es necessari indicar-los tots. De manera que en la meva crida no em feia falta introduir el paràmetre *border_condition* ja que per defecte es fals.

Un cop trobada aquesta funció que permetia obtenir un tros de la imatge, vaig crear una funció que servís per tal de poder partir una imatge en els trossos que em fes falta, com que aquesta funció havia de retornar més d'una imatge, la vaig dissenyar de manera que retornes una CimgList que contingues tots els bocins. Aquesta funció va quedar de la següent manera:

```
CImgList<float> tallar(CImg<float> imatge, int numparts){

        //obtinc l'altura de la imatge, divideixo pel numero de parts  i tallo les parts d'aquella mida.

        int x = imatge.dimx();

        int longpart= imatge.dimy()/numparts;

        int y=0;

        CImgList<float> llista;

        for(int i=0; i<numparts;i++,y+=longpart){

        llista.insert(imatge.get_crop(0,y,x,y+longpart));
```

```
        }
        return llista;
}
```

3.2.1.2 Creació d'una funció per unir imatges.

Un cop podia tallar les imatges, necessitava una funció que em permetés tornar-les a unir en una de sola. Vaig trobar la funció següent en la documentació de la llibreria Cimg.

```
CImg<T> get_append (const char axis='x', const char align = 'p') const
```

Aquesta funció permet convertir en una el conjunt d'imatges contingut en una llista. Per tant vaig crear una funció que encapsules aquesta funció de la següent manera.

```
CImg<float> ajuntar(CImgList<float> llista){

        return llista.get_append('y');

}
```

3.2.1.3 Modificació del programa en sèrie

Utilitzant aquestes funcions, que les vaig integrar en l'arxiu funcions.h de la carpeta Includes del projecte, la modificació que havia de fer en el programa en sèrie va ser la següent.

```
        // Separar els 3 canals de color RGB

        CImg <float>  red = separar_RGB(image, 0),

        green = separar_RGB(image, 1),

        blue = separar_RGB(image, 2);

        //Aplicar els difumitats

        CImg <float> bred3(red), bred4(red), bred5(red);


        //Obtinc les llistes que contenen els 5 trossos en que s'ha      partit la imatge.

        CImgList<float> llista3,llista4,llista5;

        llista3=tallar(bred3,5),llista4=tallar(bred4,5),          llista5=tallar(bred5,5);

        for(unsigned int i=0;i<llista3.size;i++){

                llista3[i].blur(3);

        }

        //un cop fet el difuminat, torno a ajuntar la imatge partida     en una sola imatge.
```

```
bred3=ajuntar(llista3);

for(unsigned int i=0;i<llista4.size;i++){

        llista4[i].blur(4);
}

bred4=ajuntar(llista4);

for(unsigned int i=0;i<llista5.size;i++){

        llista5[i].blur(5);
}

bred5=ajuntar(llista5);
//aquest procediment es repeteix per a cadascun dels canals de color
```

3.2.2 Programa en paral·lel

A causa de l'estructura del programa en paral·lel original, que era un switch on cada tasca cridava una funció diferent del programa, va sorgir el problema de com aconseguir que les funcions de difuminat poguessin ser variables, es a dir, que utilitzant el mateix executable es poguessin distribuir les funcions de difuminat de manera diferent.

En el programa original, les tasques s'identificaven definint constants i això no permetia que hi haguessin variacions en el numero de tasques sense haver d'entrar a modificar codi. A més les tasques tenien una forta dependència les unes de les altres, i aquestes s'havien de mantenir.

La solució a aquest problema va ser la creació d'un fitxer de configuració que especifiqués quina tasca havia de realitzar cada procés i llegir aquest fitxer en la clàusula default del switch, ja que si no s'especificava la tasca que havia d'executar un procés determinat la busques allí.

Aquesta solució ha permès que es pugui realitzar l'execució de l'aplicació distribuint les tasques de difuminat sense haver de modificar el codi ni crear un executable diferent.

3.2.2.1 Descripció de BASIZ amb paral·lelisme de dades.

En aquest apartat explicaré l'algoritme que utilitza el programa, tot indicant quines són les diferències entre l'algoritme nou i l'original.

Pas 1. Aquest pas consisteix en un bucle que descodifica el fitxer mpg demanat, tot obtenint imatges, un comptador controla el nombre de imatges a processar.

Per a cada imatge descodificada es passa a realitzar els següents passos:

Pas 2. Es separa la imatge en els 3 canals de color RGB

Pas 3. Es tallen les imatges obtingudes anteriorment segons els nombres desitjats per a cada nivell de difuminat, suposem que en aquest cas son 2 pel difuminat 3, 4 pel difuminat 4 i 6 pel difuminat 5. Com es pot observar en la Figura 13, com més es tallen les imatges, el trossos queden més petits. Com que després es difuminaran aquests trossos, si són massa petits la imatge pot arribar a perdre alguna part important i el programa pot marcar algunes zones diferents. Aquest pas, no apareixia en l'aplicació original ja que ha estat incorporat per poder aplicar el paral·lelisme de dades.

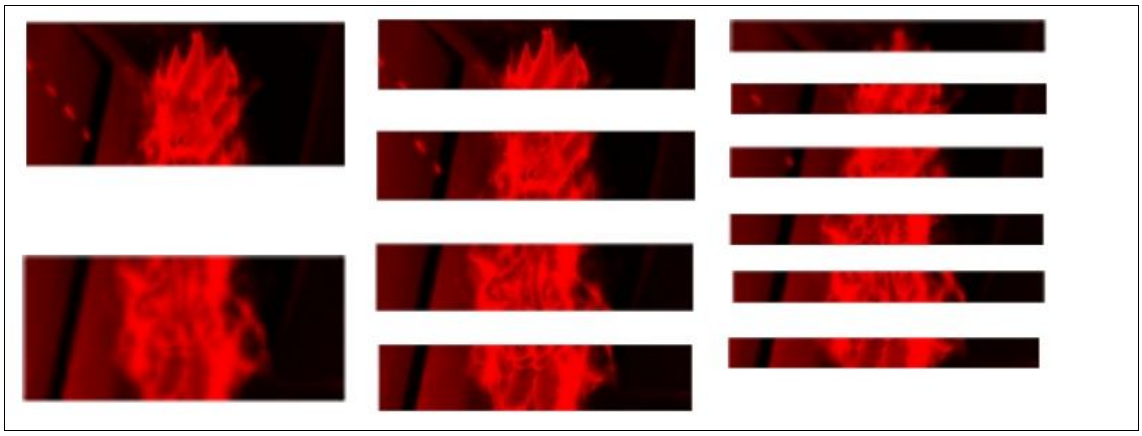

Figura 13: Exemple d'imatge tallada per realitzar els difuminats de nivell 3, 4 i 5 respectivament en 2, 4 i 6 trossos.

Pas 4. Es realitza el difuminat de les parts.

Pas 5. Com que el següent pas és sumar els diferents difuminats, les parts a sumar han de ser iguals. Com que hem dividit la imatge en diferents parts per a cada difuminat, es possible que això no sigui així. Per resoldre aquest problema, s'ajunten les parts dels difuminats que s'han separat més, obtenint així per a cada difuminat el mateix nombre de parts. Aquest és el motiu pel qual el nombre de parts a dividir les imatges han de ser múltiples entre ells. A la Figura 14 dues imatges que no es poden sumar, a la Figura 15 dues imatges que si que ho poden fer.

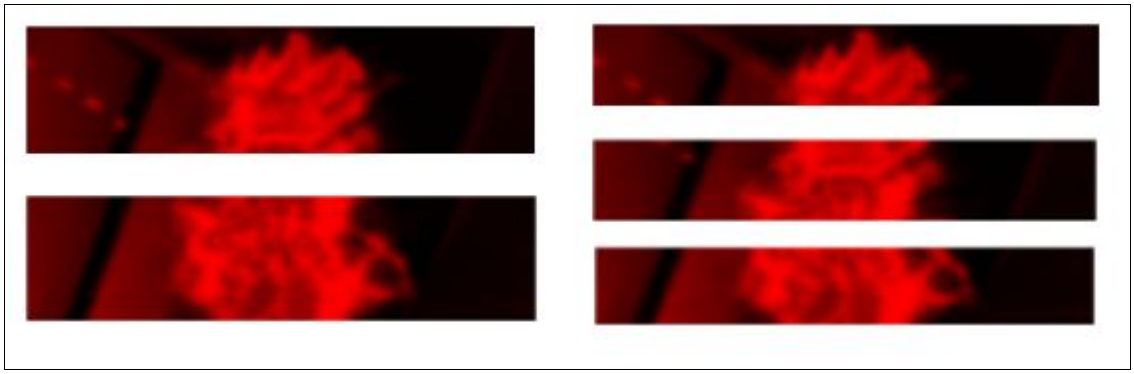

Figura 14: Imatges que no es poden sumar

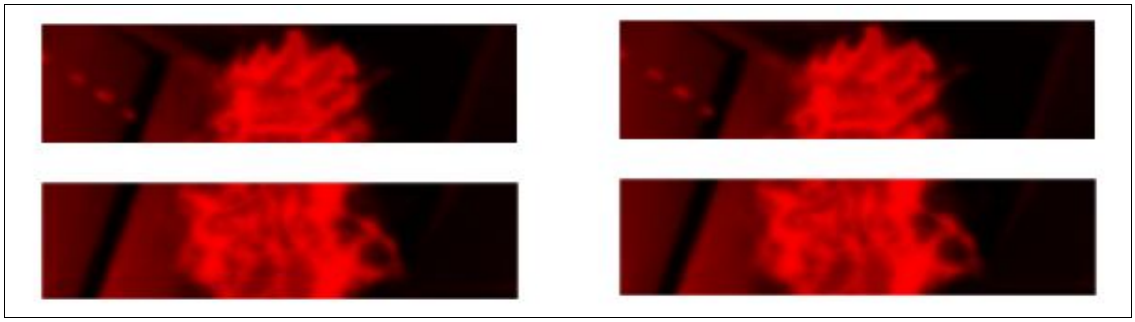

Figura 15: Imatges que si que es poden sumar

Pas 6. Es sumen les parts iguals de cada difuminat obtenint imatges difuminades de cada boci.

Pas 7. Es sumen les parts iguals de cada color. A la Figura 16 es pot observar el resultat.

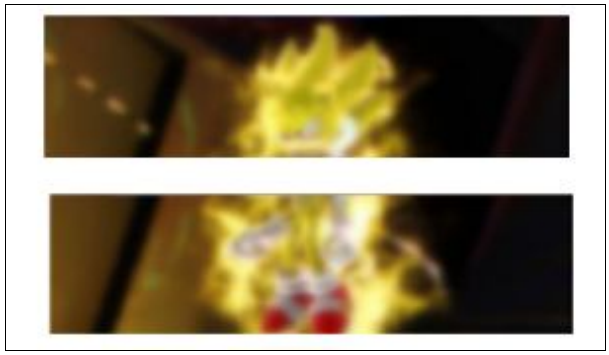

Figura 16: Imatge després de sumar els diferents colors

Pas 8. S'ajunten les diferents parts obtenint una imatge sencera amb tots els colors.

Pas 9. Es converteix la imatge en format HSB

Pas 10. Es detecten els llindars posant els píxels inferiors al llindar a negre.

Pas 11. Es marquen les zones més importants i es mostra el vídeo marcat per pantalla (Figura 17).

Figura 17: Imatge resultat del nou algoritme

En aquest nou algoritme s'hi han afegit tasques respecte l'algoritme original. Aquestes tasques són totes les referents a tallar les imatges i a ajuntar-les de nou, tot i això, hi han altres tasques que existien a l'algoritme original però que en aquest s'han clonat com són les tasques de difuminat i suma. Totes aquestes tasques han de complir les dependències que tenien en l'algoritme original.

3.2.2.2 Estructura del programa en paral·lel.

Com que el número de tasques per executar el programa és variable, depenent del grau de distribució de les tasques de difuminat, l'estructura del programa ha deixat de ser fixa, i per tant hi ha tasques que no es defineixen com a constants en l'inici del codi sinó que es llegeixen d'un fitxer.

Tot i això hi han algunes tasques que sempre són les mateixes, és a dir, que són constants encara que s'utilitzi alguna configuració diferent. Aquestes tasques estan definides com a constants igual que passava a l'aplicació original.

Així es pot dir que el programa en paral·lel te una part fixa, que es defineix en constants i una part variable que per poder-la controlar es necessita utilitzar els fitxers de configuració.

Aquesta part fixa està formada per 16 tasques que són les següents:

T0-Extreu MPG

Aquesta tasca s'encarrega d'extreure les imatges contingudes en el fitxer mpg desitjat, és igual que la funció del BASIZ original

T1,T2,T3- Separar RGB

Aquestes tasques separen un canal de color de la imatge extreta en la tasca anterior i cadascuna es guarda el canal que li toca.

T4,T7,T10- Tallar per difuminat 3

Aquestes tasques tallen cadascuna la imatge d'un dels canals en els trossos desitjats per realitzar el difuminat distribuït de nivell 3.

T5,T8,T11- Tallar per difuminat 4

Aquestes tasques tallen cadascuna la imatge d'un dels canals en els trossos desitjats per realitzar el difuminat distribuït de nivell 4

T6,T9,T12- Tallar per difuminat 5

Aquestes tasques tallen cadascuna la imatge d'un dels canals en els trossos desitjats per realitzar el difuminat distribuït de nivell 5

T13- Ajuntar tot

Aquesta tasca es la tasca que ajunta els trossos d'imatge que ja han estat difuminats, no es la única tasca de suma ja que també n'hi ha que està distribuïda.

T14- Converteix a HSB

Aquesta tasca converteix la imatge difuminada en HSB per tal que l'algoritme sigui capaç de detectar les zones més importants.

T15- Calcula el llindar

Aquesta tasca calcula el llindar per poder decidir quins píxels son significatius i quins no.

T16- Detecta zones importants

Aquesta tasca detecta les zones importants basant-se en els llindars calculats anteriorment i marca les zones més importants en la imatge original extreta del fitxer per així mostrar-la per pantalla.

Aquestes 16 tasques son les que s'executen sempre, sigui quina sigui la seva configuració. La resta de tasques es situarien entre la T12 i la T13 en l'algoritme. Aquestes 16 tasques tenen una entrada en l'switch principal del programa, la resta de tasques es resolen en la clàusula default.

La Figura 18 mostra com s'estructuren les tasques per a 1 imatge pel difuminat de nivell 3 i 2 pels difuminats de nivell 4 i 5:

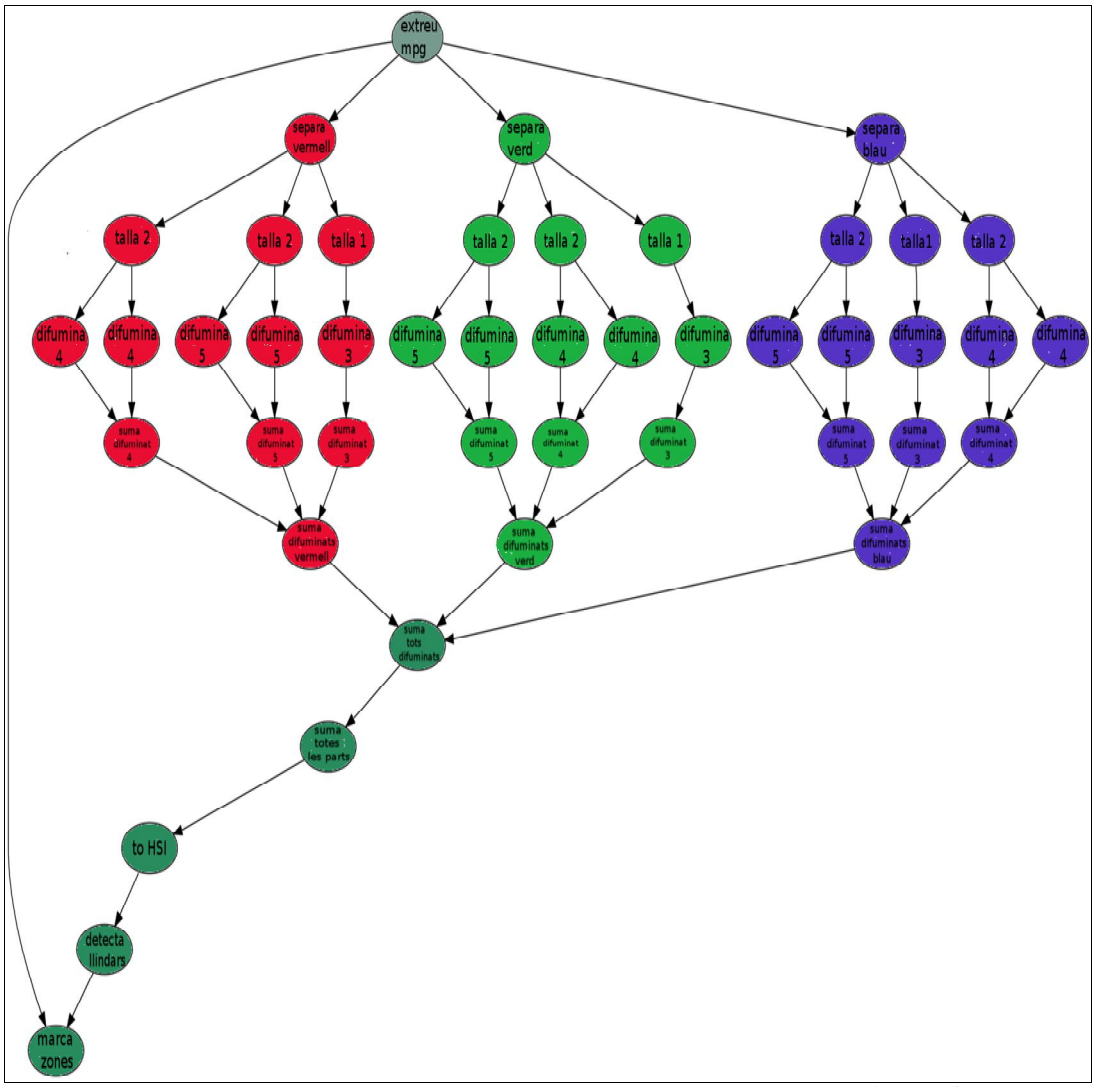

Figura 18:Imatge on es mostra l'esquema per una determinada configuració

3.2.2.3 Fitxer de configuració

Els programes que utilitzen MPI identifiquen cada procés mitjançant un número que serveix per poder-los diferenciar i fer que cadascun realitzi tasques diferents.

En el programa original de BASIZ, hi havia un switch que segons aquest número provocava l'execució de una part de l'algoritme. El problema era que en la nova versió la tasca que havia d'executar cada procés depenia del nombre de parts en que es volien dividir les imatges.

Arran d'això, vam decidir l'elaboració d'un fitxer de configuració de tal manera que contingues la informació necessària perquè cada procés executés la tasca que li corresponia.

Aquest fitxer és un fitxer en format de text pla i que té el format com s'explica a continuació.

El fitxer comença amb un registre que guarda el mínim de parts en que es vol dividir la imatge per realitzar els diferents difuminats. Per exemple, si es vol dividir la imatge en sis trossos pel difuminat 3, un pel difuminat 4 i 3 pel difuminat 5, el valor que es guarda en aquest registre es 1.

Tot seguit el fitxer conté el nombre de divisions que es desitja per a cada tipus de difuminat, aquests valors han de ser múltiples del nombre mínim. Aquests registres seran de la següent forma:

<identificador> <valor>

On identificador pot tenir els valors MI per indicar el registre amb el mínim, B3 per indicar el de difuminat 3, B4 pel difuminat 4 i B5 pel difuminat 5.

Els següents registres indiquen a cada tasca la part de l'algoritme que ha d'executar de forma que el primer camp de l'entrada és el identificador de tasca, el segon és la tasca a executar, el tercer és el identificador del procés origen i l'últim l'identificador del procés destí. Així una entrada del fitxer de configuració qualsevol seria d'aquesta forma:

<id_tasca> <id_funció> <id_tasca_origen> <id_tasca_destí>

Així els fitxers de configuració tindran aquest format, en aquest exemple només se'n mostra una part ja que es molt extens:

```
MI 3
B3 3
B4 6
B5 9

17 BR3 4 71
18 BR3 4 72
19 BR3 4 73

20 BR4 5 74
21 BR4 5 74
22 BR4 5 75
23 BR4 5 75
24 BR4 5 76
25 BR4 5 76

26 BR5 6 77
27 BR5 6 77
28 BR5 6 77
29 BR5 6 78
30 BR5 6 78
31 BR5 6 78
32 BR5 6 79
33 BR5 6 79
34 BR5 6 79

35 BG3 7 80
36 BG3 7 81
37 BG3 7 82

38 BG4 8 83
39 BG4 8 83
```

El primer registre del fitxer, MI 3 indica que el nombre de divisions mínim que es fa en una imatge per a un difuminat és 3. Els altres nombres de divisions han de ser múltiples d'aquest.

Els següents registres indiquen la quantitat de divisions per a cada difuminat. B3 3 indica que per al difuminat 3 hi hauran 3 divisions, B4 6 indica que pel difuminat 4 n'hi hauran 6 i B5 9 indica que pel difuminat 5 hi hauran 9 divisions.

La resta de registres indiquen a cada tasca quina operació ha de realitza i quines son les seves tasques origen i destí. Per exemple el registre 17 BR3 4 71 indica que la tasca 17 realitzarà el difuminat 3 a la imatge rebuda de la tasca 4 i enviarà el resultat a la tasca 71.

Si el fitxer està ben generat, cada tasca buscarà el seu identificador i un cop trobat, tindrà tota la informació que necessita per executar la seva tasca.

Generar aquest fitxer manualment es complicat ja que cada tasca es fortament dependent de la configuració i si es realitza algun error, el programa fallaria.

Per exemple si una tasca envia la informació a una altra tasca, però aquesta no espera rebre informació de la primera, l'aplicació quedarà bloquejada.

Ja que la realització d'aquest fitxer es molt complicada, hem dissenyat un programa que genera els fitxers de configuració només rebent el número de parts en que es vol distribuir cada tipus de difuminat.

Aquest petit programa, a més de generar el fitxer de configuració també indica la manera com s'ha d'executar l'aplicació en paral·lel, ja que en executar un programa amb MPI s'ha de saber el nombre de processos que hi ha, i aquests varien depenent de la configuració que s'utilitzi.

3.2.2.4 Modificacions en el codi.

Per tal de tallar i ajuntar les imatges en el programa en MPI, he hagut de definir i implementar les funcions corresponents per tal que els processos encarregats a fer aquestes tasques, ho poguessin fer.

Per fer aquestes funcions, he seguit l'estructura de les funcions destinades a realitzar les altres tasques. Aquesta estructura consisteix en que el procés crida la funció un cop sap les tasques origen i destí, que són alguns dels valors passats per paràmetre.

Un cop cridada la funció, aquesta en primer lloc espera rebre la imatge de la tasca origen, després executa la operació pertinent i un cop fet torna a enviar les dades a la tasca destí.

A continuació s'especifica el codi que correspon a la funció que talla les imatges.

```
int t_tallar(int times, int origen, int desti, int parts) {

        CImg<unsigned char> tmp_img;

        CImgList<unsigned char> tmp_list;
```

```
        im_frame * frame;
        int tamany=0;
        while (times) {
                t_inici_espera = MPI_Wtime();
                rebre_imatge(frame, origen);
                t_espera += MPI_Wtime() - t_inici_espera;
                tmp_img.load_mem_ppm(*frame);
                free(frame->buffer);
                free(frame);
                tmp_list=tallar(tmp_img,parts);
                for(unsigned int i=0;i<tmp_list.size;i++){
                        tmp_list[i].save_mem_ppm(frame);
                        tamany=enviar_imatge(frame, desti+i);
                        free(frame->buffer);
                        free(frame);
                }
                t_tamany_enviat+=tamany;
                times--;
    }
        return 0;
}
```

Les funcions rebre imatge i enviar imatge, són funcions que encapsulen les funcions de MPI_Send i MPI_Recv. Com que aquestes funcions envien cadenes de bytes, s'han de convertir les imatges en frames per ser enviades, i un cop es rep un frame, convertir-lo en l'objecte Cimg.

Aquestes conversions es duen a terme gràcies a les funcions save_mem_ppm i load_mem_ppm.

Per ajuntar les imatges de nou vaig dissenyar la següent funció.

```
int t_ajuntar(int times, int origen, int desti, int parts) {
```

```
        CImg<unsigned char> tmp_img;
        CImgList<unsigned char> tmp_list;
        im_frame * frame;
        while (times) {
                for(int i=0;i<parts;i++){
                        t_inici_espera = MPI_Wtime();
                        rebre_imatge(frame, origen+i);
                        t_espera += MPI_Wtime() - t_inici_espera;
                        tmp_img.load_mem_ppm(*frame);
                        free(frame->buffer);
                        free(frame);
                        tmp_list.insert(tmp_img);
                }
                tmp_img=ajuntar(tmp_list);
                tmp_list.clear();
                tmp_img.save_mem_ppm(frame);
                t_tamany_enviat+=enviar_imatge(frame, desti);
                free(frame->buffer);
                free(frame);
                times--;
    }
        return 0;
}
```

Per a que cada procés executes la seva funció, vaig escriure la clàusula default del switch de la següent manera:

```
default:
//aquest bucle fa que cada procés busqui el seu registre al fitxer.
```

```
while(idtask!=id_tasca){

        fscanf(confile,"%d%s%d%d",&idtask,&nomtask,&origen,&desti);

}

if(strcmp(nomtask,"BR3")==0 || strcmp(nomtask,"BG3")==0 ||
strcmp(nomtask,"BB3")==0){

        t_blur(times,origen,desti,3);

}

if(strcmp(nomtask,"BR4")==0 || strcmp(nomtask,"BG4")==0 ||
strcmp(nomtask,"BB4")==0){

        t_blur(times,origen,desti,4);

}

if(strcmp(nomtask,"BR5")==0 || strcmp(nomtask,"BG5")==0 ||
strcmp(nomtask,"BB5")==0){

        t_blur(times,origen,desti,5);

}

if(strcmp(nomtask,"AR3")==0 || strcmp(nomtask,"AG3")==0 ||
strcmp(nomtask,"AB3")==0){

        parts=b3/min;

        t_ajuntar(times,origen,desti,parts);

}

if(strcmp(nomtask,"AR4")==0 || strcmp(nomtask,"AG4")==0 ||
strcmp(nomtask,"AB4")==0){

        parts=b4/min;

        t_ajuntar(times,origen,desti,parts);

}

if(strcmp(nomtask,"AR5")==0 || strcmp(nomtask,"AG5")==0 ||
strcmp(nomtask,"AB5")==0){

        parts=b5/min;

        t_ajuntar(times,origen,desti,parts);

}

if(strcmp(nomtask,"SDR")==0 || strcmp(nomtask,"SDG")==0 ||
strcmp(nomtask,"SDB")==0){
```

```
                t_add(times,origen,desti,min);
        }
        if(strcmp(nomtask,"STC")==0){
                t_add(times,origen,desti,min);
        }
        break;
```

3.2.3 Generador de XML per calcular els mappings

Per si alguna vegada es vol provar aquesta aplicació en un clúster, la gran quantitat de processos que hi apareixen fa difícil l'elecció d'un mapping correcte. Sabent l'existència de programes destinats a calcular mappings basant-se en un fitxer XML, he generat un petit programa que rebent dades de l'execució del programa en paral·lel i el fitxer de configuració genera el fitxer XML necessari per calcular els mappings.

El format del fitxer en XML necessari és de la següent forma:

```
<?xml version=\"1.0\" ?>
        <program name=\"basiz\">
                <task id="1">
                        <for count="50">
                                <receive id="0" task="16"/>
                                <exec run="1200" />
                                <send id="0" task="17" vol="2500" />
                        </for>
                </task>
        </program>
```

-L'etiqueta task es repeteix per totes les tasques de l'aplicació.
-Les etiquetes receive i send es repeteixen tantes vegades com faci falta.
-El valor de run de la etiqueta exec es el temps en microsegon.
-El valor de vol en la etiqueta send es el tamany a enviar en bytes.

Per tant, per realitzar el fitxer en XML, el programa requereix un fitxer amb les dades de sortida al finalitzar l'execució del programa en paral·lel.

També es necessita el fitxer de configuració corresponent per tal de poder obtenir les relacions entre les diferents tasques.

El codi principal del generador del fitxer XML és el següent:

```
        fprintf(xmlfile,\t<task id=\"%d\">\n",idtaska);
```

```
fprintf(xmlfile,"\t\t<for count=\"%d\">\n",imatges);

for(int i=0;i<receive.size();i++){

        fprintf(xmlfile,"\t\t\t<receive id=\"0\" task=\"%d\"/>\n",receive[i]);

}

fprintf(xmlfile,"\t\t\t<exec run=\"%d\"/>\n",
(int)((temps/imatges)*(float)1000000));

for(int i=0;i<send.size();i++){

        fprintf(xmlfile,"\t\t\t<send id=\"0\" task=\"%d\" vol=\"%d\"/>\n", send[i],vol/imatges);

}

fprintf(xmlfile,"\t\t</for>\n");

fprintf(xmlfile,"\t</task>\n");
```

Ja que realitzava aquest programa per crear un fitxer XML per calcular els mappings, també vaig incorporar-hi la generació d'un fitxer en format dot, que permet generar gràfics per tal de poder observar com es distribueix l'aplicació.

El format dot es un format que permet generar gràfics de forma senzilla. Per generar un gràfic senzill n'hi ha prou amb escriure el següent:

```
digraph A {
        A -> B
        B -> C
        B -> D
}
```

un cop es té el fitxer en format .dot, generar el gràfic es tan senzill com:

-Instal·lar el paquet Grahpviz en cas que no estigui instal·lat.
-Escriure en una terminal: dot -Tpng fitxer.dot -o grafic.png

Un cop fet això apareix el fitxer grafic.png que conté un gràfic com el de la Figura 19.

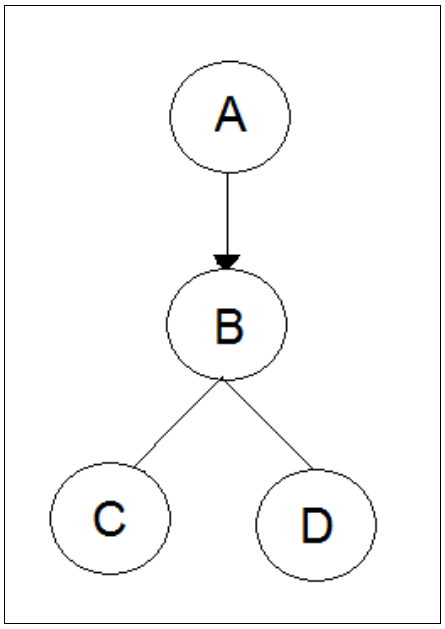

Figura 19: Gràfic resultant de l'arxiu .dot anterior

4- Preparació de l'entorn

4.1 Instal·lació

Per configurar un ordinador amb ubuntu i poder executar i compilar el programa s'han d'instal·lar els paquets següents:

Paquets necessaris per compilar.

-libxrandr-dev

-libX11-dev

-libxext-dev

-imagemagick
-g++
-gcc

Paquets necessaris per la compilació i execució en MPI

-libmpich1.0gf
-libmpich1.0-dev
-mpich-bin

Aquests paquets es poden instal·lar sense cap problema utilitzant els repositoris del sistema de l'ubuntu:

sudo apt-get install <nom del paquet>

4.2 Compilació i execució

Per compilar el projecte, un cop instal·lats els paquets anteriors, utilitzem el make que hi ha dins la carpeta arrel del projecte.

Un cop compilat el projecte, apareixen sis executables: els dos programes en sèrie (l'original i el nou), els dos en paral·lel, el generador del fitxer de configuració i el generadors de XML i dot.

Els quatre primers executables realitzen el mateix algoritme, que es l'algoritme del BASIZ, la diferència es que els programes en sèrie realitzen l'algoritme seqüencialment sense necessitat d'utilitzar el MPI, i els paral·lels utilitzen MPI per tal de realitzar el tractament de les imatges, tot definit un seguit de tasques que s'encarreguen de petites parts del algoritme i que es poden executar paral·lelament.

Per tal d'executar el programa en sèrie original, només cal passar per paràmetre el arxiu de vídeo que volem que tracti, així hauríem d'utilitzar:

./progSERIEOriginal <arxiu de vídeo>

De la mateixa manera per executar el programa en sèrie nou s'ha d'utilitzar:

./progSERIE <arxiu de vídeo>

Pel programa en paral·lel original, hi han dues maneres d'executar-lo, depenent de si volem utilitzar un fitxer de maping (fitxer on s'indica quines tasques s'han d'executar a quina màquina) o no, en cas de no utilitzar un fitxer de mapping s'ha d'indicar el numero de tasques necessàries per la realització del algoritme, en el cas del MPI son 20.

El format d'un fitxer de mapping consta d'una entrada per a cada tasca, aquestes entrades han de contenir la següent informació.

<nom màquina> <distància de la maquina origen> <programa a executar>

En els dos casos, s'ha de indicar per paràmetre l'arxiu de vídeo a utilitzar i el numero de imatges per ser tractades.

Així doncs per executar sense fitxer de mapping quedaria:

mpirun -np 20 ./progMPI <arxiu de vídeo> <numero imatges>

i amb fitxer de mapping:

mpirun -p4pg <arxiu_mapping> <arxiu vídeo> <numero imatges>

Per tal d'executar el programa en paral·lel nou, el primer que cal fer es crear un fitxer de configuració.

El podem crear fàcilment utilitzant:

./config

I contestant les preguntes sobre les parts per a cada difuminat.

Un cop creat el mateix programa ens indica quina es la forma adequada per executar l'aplicació tot indicant-nos quin nombre de tasques s'ha d'utilitzar. Així la crida per executar serà de la forma:

mpirun –np <x> ./progMPI <arxiu vídeo> <numero imatges> < fitxer configuració>

Com en el programa original, també es pot utilitzar un fitxer de mapping per tal de executar el programa en paral·lel:

mpirun -p4pg <arxiu_mapping> < vídeo> <numero imatges> <configuració>

Si es disposa d'un generador de mapings a partir d'un fitxer XML es pot generar aquest fitxer de la següent manera:

./gen <fitxer configuració> <fitxer dades> <fitxer XML(nou)> <fitxer .dot(nou)> <numero imatges>

5- Experimentació

Quan el programa va estar fet i en funcionament, va arribar el moment de realitzar tot un seguit de proves i comparar els resultats.

S'han fet dos tipus de proves, unes per determinar si s'ha millorat el rendiment en les tasques de difuminat, i unes altres per observar si l'aplicació es comporta de la mateixa manera tot i aplicar paral·lelisme de dades.

En el primer tipus de proves, s'ha executat l'aplicació original i diverses configuracions diferents, i s'han analitzat els temps d'execució de les tasques de difuminat.

En el segon tipus s'ha executat l'aplicació i s'han observat les zones que eren marcades com a importants comparant-les amb les que marcava l'aplicació original.

5.1 Proves de rendiment

El primer que cal fer, ja que més endavant es voldrà millorar el rendiment, es mostrar una gràfica amb el temps que tarda cadascuna de les tasques que executa BASIZ (Figura 20). Com que els processos que executen les mateixes tasques tenen còmputs semblants, en els següents gràfics només apareix 1 tasca de cada tipus.

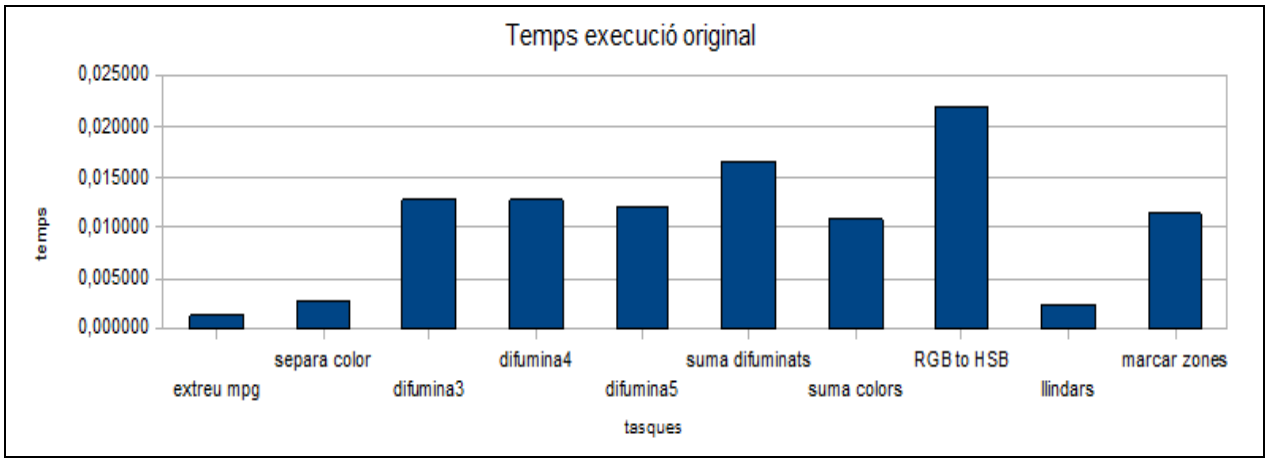

Figura 20: Gràfica de temps d'execució del BASIZ original

Es pot observar que la tasca que tarda més temps correspon a la conversió de RGB a HSB. Quan es vulgui augmentar el rendiment de l'aplicació, aquesta serà la tasca que s'haurà de dividir ja que es la que té un còmput més elevat.

Per parlar de diferents configuracions utilitzarem el conveni següent.

Una configuració n1-n2-n3 es aquella que es divideix la imatge en n1 parts pel difuminat 3, n2 pel difuminat 4 i n3 pel difuminat 5.

Si observem el còmput de les tasques per a una configuració 2-2-2 observarem que tot i que les tasques de difuminat han millorat el seu rendiment, la tasca amb més pes continua sent la de conversió a HSB (Figura 21).

Igual que abans, només es mostra una barra per a cada tipus de tasca ja que els processos que realitzen tasques similars tenen el mateix còmput.

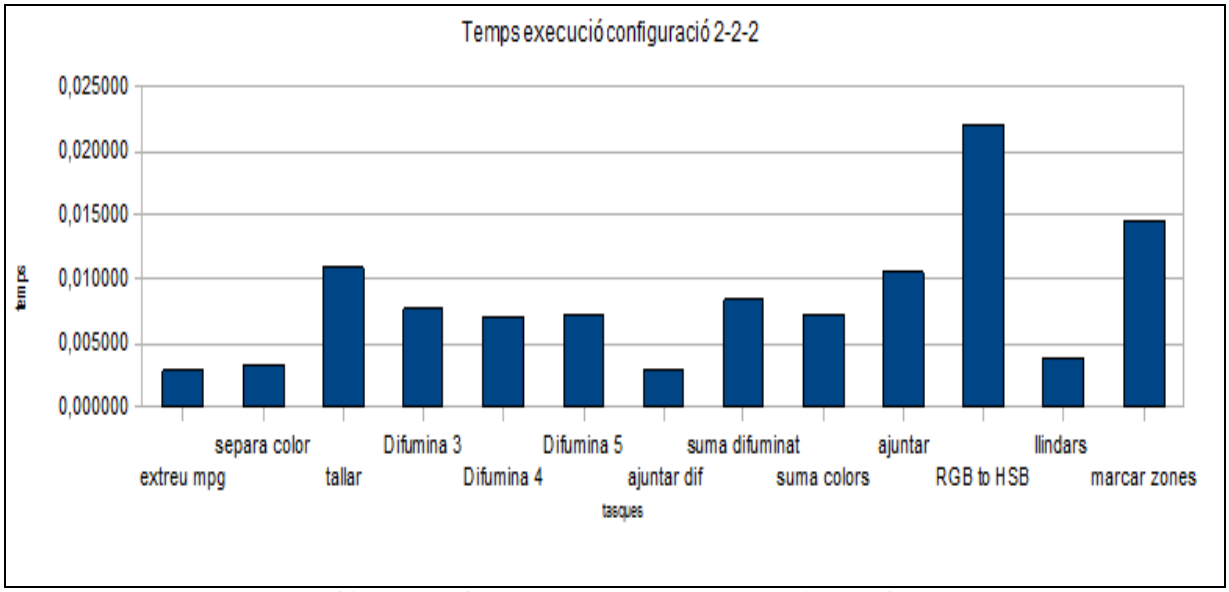

Figura 21: Gràfic amb el còmput de les tasques per una configuració 2-2-2

Ja que en aquest projecte hem dividit les tasques de difuminat, tot seguit veurem com millora el rendiment d'aquestes tasques per a diferents configuracions. En la gràfica de la Figura 22 es veu com evoluciona el rendiment de les tasques de difuminat de cada nivell a mesura que es fan més divisions.

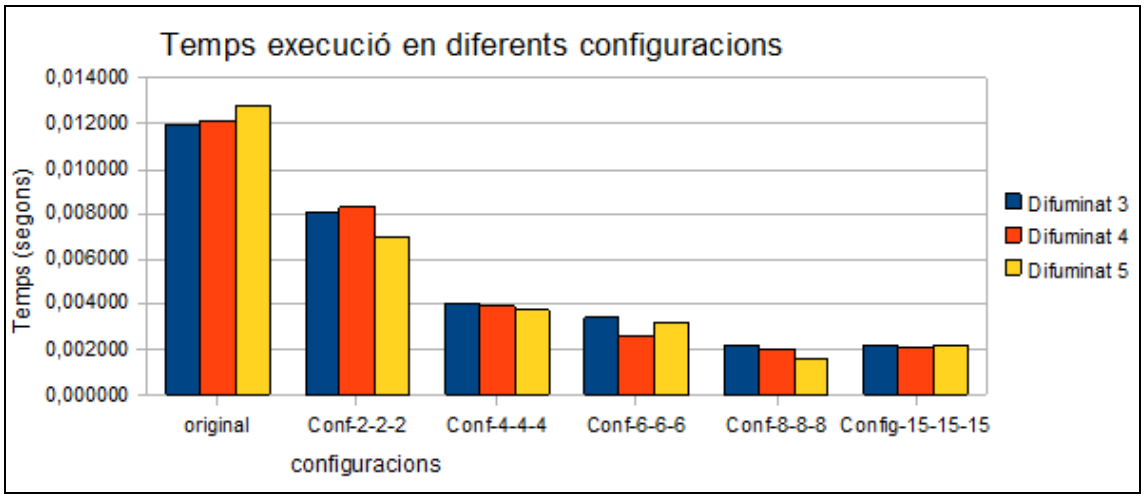

Figura 22: Gràfic on s'observa la millora de rendiment per diferents configuracions

Es pot observar que a partir d'un nombre de divisions, el guany obtingut en rendiment s'estabilitza i per tant no cal dividir en moltes parts, tarda el mateix dividir per 8 que per

15.

5.2 Proves de verificació de resultats

Un cop vist això anem a determinar fins a quin punt es pot dividir per a que les imatges detectin les mateixes zones.

Per fer aquestes proves, s'ha executat l'aplicació original de BASIZ i s'ha escollit una imatge del vídeo per tal d'observar les zones marcades.

Després s'ha executat el programa amb les configuracions anteriors observant el resultat en la imatge escollida.

Comparant la imatge original amb les obtingudes utilitzant diferents configuracions es pot observar que si hi han poques parts, les imatges son semblants (Figura 23).

Figura 23: Configuració amb resultats semblants a l'original

En canvi amb moltes divisions, el resultat es bastant diferent.(Figura 24).

Figura 24: Imatge amb resultats diferents de l'original

Així doncs com a resultat final jo diria que amb una configuració de 4-4-4 la imatge es

bastant fiable, tot i que poden aparèixer zones noves, i el rendiment augmenta considerablement.

Tot i amb això crec que ha de ser l'usuari qui decideixi quina es per ell la millor configuració depenent del nivell de deformacions que està disposat a tolerar i el rendiment que necessita assolir.

6- Conclusions

La realització d'aquest treball ha consistit en l'aplicació del paradigma de paral·lelisme de dades a un dels passos de l'aplicació BASIZ (Bright And Saturated Image Zones).

BASIZ és una aplicació de tractament d'imatges digitals que consta d'un conjunt de passos, de tal manera que finalment detecta les zones amb més lluminositat i intensitat de color de cada imatge pertanyent a un vídeo en format mpg.

En concret s'ha aplicat el paral·lelisme de dades a les tasques que implementen els diferents nivells de difuminat de cada imatge.

La implementació del paral·lelisme de dades ha consistit en separar les imatges en el numero de trossos que es desitgi i aplicar l'algoritme de difuminat sobre cadascun d'aquests trossos per separat.

Un cop obtingudes les imatges difuminades, aprofitant que estan separades, també s'han aplicat les tasques de suma de forma paral·lela.

Un cop s'ha obtingut el programa funcionant correctament s'ha realitzat un procés d'experimentació per analitzar les millores en el rendiment de les tasques distribuïdes i comprovar si els resultats obtinguts són iguals als resultats originals.

S'ha observat que els resultats varien a mesura que es va augmentant el nombre de parts. Per tant si es realitzes un nombre elevat de divisions la imatge podria resultar bastant deformada. Per contrapartida a partir d'un nombre de divisions, el rendiment s'estabilitza i ja no augmenta.

Per tant és pot arribar a la conclusió en un nombre reduït de divisions l'aplicació de paral·lelisme de dades pot suposar una millora en el rendiment de l'aplicació.

Treball Futur:

Com a continuació del treball realitzat ens plantegem aplicar paral·lelisme de dades a la tasca de conversió de RGB a HSB. Ja que aquesta tasca es la que suposa un còmput més alt en la realització de l'algoritme i es la tasca que fa perdre rendiment a l'aplicació BASIZ.

Per tant un cop s'ha vist que el paral·lelisme de dades es pot aplicar a aquesta aplicació, el següent pas seria paral·lelitzar aquesta tasca.

7- Bibliografia

[1] David E.Culler. Jaswinder P. Singh. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers. 1999.

[2] Message Passing Interface Forum. MPI: A message-passing interface standar. Journal of Supercomputer Applications. Vol. 8. Num. 3/4. 1994.

[3] Javier Borrás. Desarrollo de una aplicación paralela con pralelismo funcional: detección basada en la visión humana. Projecte Fi de Carrera. Departament d'Informàtica. Universitat Autònoma de Barcelona. 2001.

[4] A. Geist, A. Beguelin, J. Dongarra , W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing, The MIT Press. Cambridge, Massachussets. 1994.

[5] Jordi Píriz. Implementació de BASIZ amb MPI. Treball Final de Carrera. Escola Politècnica Superior.Universitat de Lleida 2008.