RICERCA DEL NUMERO DI CLUSTER DI UN DATASET RISPETTO UNA GRIGLIA DI PUNTI N-DIMENSIONALE

Il progetto mostra una possibile soluzione di algoritmi volti a identificare il numero di cluster presenti all'interno di un certo dataset n-dimensionale

Matteo Di Fraia
difria.matteo@gmail.com

Sommario

INTRODUZIONE

Il progetto svolto è la prosecuzione del lavoro del collega Giuseppe Raimondi: si parte dalla *membership function* creata dal precedente lavoro, e da questa tramite una serie di operazioni viene plottato il cluster che ragionevolmente[1] rispecchia i dati in ingresso.

Questa serie di operazioni parte dalla *membership function* e prevede la creazione di 100 immagini binarie in funzione di 100 valori di soglia; per ognuna di queste, viene rintracciato il numero di cluster presenti, assegnando ad ogni '1' binario un label, successivamente creato il dendrogramma che rispecchia la gerarchia di trasformazione dei vari cluster trovati e identificata l'immagine che meglio rispecchia i cluster del dataset osservando il ramo del dendrogramma che si mantiene costante maggiormente rispetto tutti gli altri.

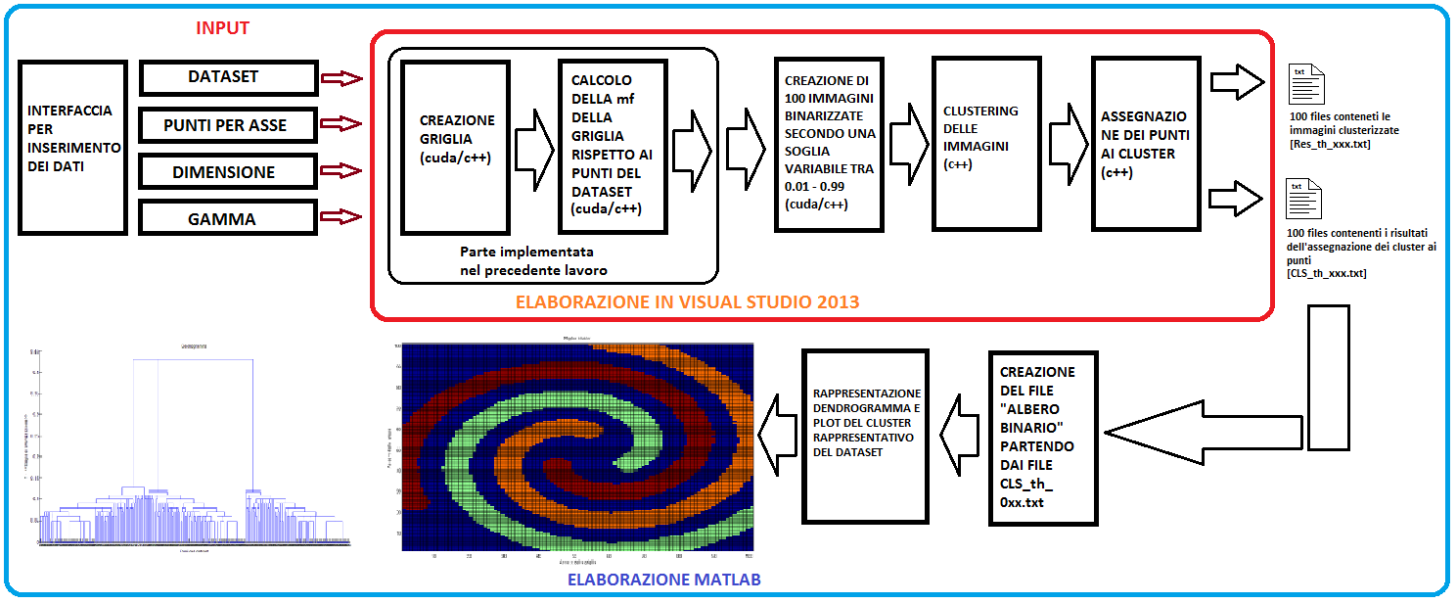
[1] Il termine "ragionevolmente" sarà chiaro a breve.

DESCRIZIONE DEL PROGETTO

Il progetto si è articolato in due parti: la prima nel programma di sviluppo Visual Studio 2013 in linguaggio C++/CUDA, e una seconda parte di elaborazione dati in Matlab.

L'elaborazione in Visual Studio si è svolta con tecnologia CUDA solamente per il calcolo delle varie immagini binarie create, mentre l'assegnazione delle etichette ai vari punti binari è stata realizzata serialmente in C++: questa scelta è stata obbligata dalla intrinseca serialità dell'algoritmo di clustering che non permette, sebbene numerose strade intraprese, la parallelizzazione del problema.

Lo schema riportato in figura riassume cosa e dove è stato fatto durante il progetto, per fornire una più chiara comprensione di quanto ci si appresta a descrivere.



La struttura dati in ingresso al progetto, messa a disposizione dal lavoro del collega, prevede una matrice N *x N x (dim - 1)* punti, che rappresenta una griglia di punti[2] i cui valori variano tra 0 ed 1: questi valori indicano l'appartenenza di un punto del *dataset* ad un punto della griglia.

DESCRIZIONE DEI BLOCCHI COSTITUTIVI – Visual Studio

Si analizzano adesso le varie funzioni richiamate all'interno del progetto Visual Studio, le quali rappresentano ognuna una parte di elaborazione schematizzata nella precedente immagine.

Binarizzazione

La prima parte di binarizzazione, prevede la forzatura del valore della griglia a 0 o 1 in funzione di una soglia; questa operazione è stata eseguita per tutti i valori di th compresi tra 0.01 e 0.99. La funzione utilizzata nel codice è la

```
binarize << <(int)pow((double)n, (double)dim - 1), n >> > (dev_bf, dim, n, th, end_index);
```

[2] Una matrice di dati *double*.

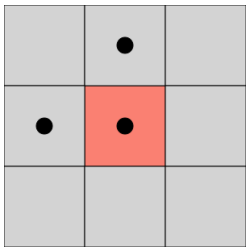
a cui vengono passati i valori della *mf*, dimensione, risoluzione della griglia soglia e massimo indice di spiazzamento dei dati nella struttura `dev_bf`. Questa viene eseguita su scheda grafica Nvidia cuda chiamando n^{dim-1} blocchi da n threads ciascuno. Ogni thread non farà altro che confrontare il valore della griglia nella posizione identificata dal suo ID con la soglia e eseguire la scrittura. Il codice implementato è il seguente:

```
__global__ void binarize(double* mf, int dim, int n, double th, int end_index){
	int j = threadIdx.x;
	int i = blockIdx.x;
	int index = i*n + j;
	if (index < end_index){
		if (mf[index] < th) mf[index] = 0;
		else mf[index] = 1;
	}
}
```

Ricerca delle componenti connesse: assegnazione dei cluster

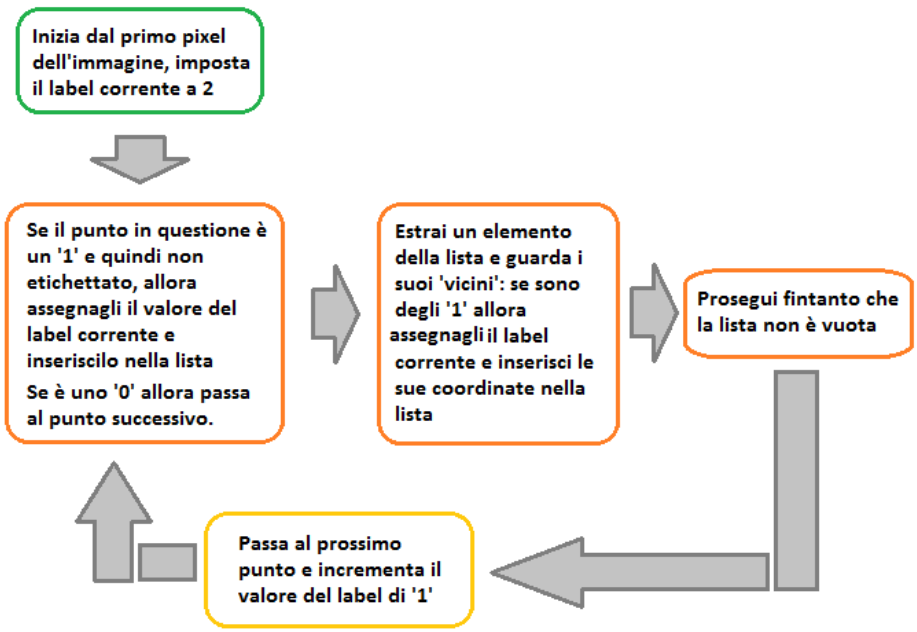
Una volta ottenuta l'immagine binarizzata in funzione di th, viene chiamata la funzione

```
findCluster(double* bf, int dim, int n);
```



Eseguita serialmente in c++, prevede di rintracciare i cluster all'interno dell'immagine binaria assegnando un etichetta che banalmente sarà un valore sempre crescente ≥2. La ricerca dei punti di contatto è eseguita secondo un criterio a 4 punti, cercando cioè la connettività delle componenti nella 4 direzioni ortogonali[3].

L'algoritmo utilizzato prevede di etichettare tutti i punti di uno stesso cluster alla volta, prima di passare ad analizzare altri punti della griglia. Il comportamento può essere riassunto dal seguente schema:



[3] In 3 dimensioni o più la ricerca coinvolge 6 punti di contatto.

Per fare questo è stata creata una struttura dati a lista concatenata e le funzione per eseguire le operazioni di push e pop:

```
struct nodo{
	int val;
	nodo* next;
};

int pop(nodo*& list){
	//ritorna il valore del nodo in cima e lo cancella
	int temp = 0;
	temp = list->val;
	nodo* q = list;
	list = list->next;
	delete q;
	return temp;
}

void push(int index, nodo*& list){
	//aggiunge un nuovo nodo in cima alla lista
	nodo* q = new  nodo;
	q->val = index;
	q->next = list;
	list = q;
	return;
}
```

Il codice che esegue l'algoritmo è il seguente:

```
void findCluster(double* bf, int dim, int n){
	//versione seriale dell'algoritmo di clustering
int i = 0;
int j = 0;
int temp_index;
int index = i*n + j;
int currlabel = 1;  //variabile per fare il label dei cluster
nodo* list = new nodo;
list = NULL;
for (i = 0; i < (int)pow((double)n, (double)(dim - 1)); i++){
	//scorre tutte le "righe" della matrice di punti (se consideriamo l'esempio 2d)
	for (j = 0; j < n; j++){
		//scorre tutte le colonne (sempre considerando l'esempio 2d)
		index = i*n + j;
		if (bf[index] == 1){
			//pixel non ancora labelato
			currlabel++;
			bf[index] = currlabel;
			push(index, list);
		}while (list != NULL){
			temp_index = pop(list);
			if ((temp_index + 1) < (int)pow((double)n, dim) && bf[temp_index + 1]
== 1 && (((temp_index) % n) < ((temp_index + 1) % n))) {
				//la terza condizione if si assicura che non prende cluster
adiacenti ma ai 2 confini opposti(x=0,X=n)
				bf[temp_index + 1] = currlabel;
				push(temp_index + 1, list);
			}
			if ((temp_index + n) < (int)pow((double)n, dim) && bf[temp_index + n]
== 1){
				bf[temp_index + n] = currlabel;
				push(temp_index + n, list);
			}
			if ((temp_index - 1) >= 0 && bf[temp_index - 1] == 1 &&
(((temp_index) % n) > ((temp_index - 1) % n))){
				//la terza condizione if si assicura che non prende cluster
adiacenti ma ai 2 confini opposti(x=0,X=n)
				bf[temp_index - 1] = currlabel;
				push(temp_index - 1, list);
			}
```

```
                    if ((temp_index - n) >= 0 && bf[temp_index - n] == 1){
                           bf[temp_index - n] = currlabel;
                           push(temp_index - n, list);
                    }
                    if (n > 2){
                    //se siamo in più di due dimensoni, allora devo controllare le altre
facce dell'ipercubo, che si troveranno spiazzati di n^2 rispetto l'indice
                           int next_dim = n*n;
                           //facciata nella dimensione maggiore
                           if ((temp_index + next_dim) < (int)pow((double)n, dim) &&
bf[temp_index + next_dim] == 1) {
                                  bf[temp_index + next_dim] = currlabel;
                                  push(temp_index + next_dim, list);
                           }
                           //facciata nella dimensione inferiore
                           if ((temp_index - next_dim) >= 0 && bf[temp_index - next_dim]
== 1){
                                  bf[temp_index - next_dim] = currlabel;
                                  push(temp_index - next_dim, list);
                           }
                    }
             }
       }
}
}
```

Assegnazione dei punti ai cluster

Questa parte del programma si occupa di assegnare, per la data soglia di binarizzazione usata, ogni punto del dataset ad un cluster. La funzione che esegue questa operazione è:

```
void assignPointsToCluster(double* cf, int dim, int n, double* points, int patterns, tab*
data)
```

Per il funzionamento è necessaria un'altra struttura dati concatenata e una funzione per la sua creazione:

```
struct tab{
       int point_numbers;
       int* cluster;
       double th;
       bool zero;
       tab* next_th;
};

void new_tab(int point_n, double _th, tab*& data){
       tab* q = new tab;
       q->point_numbers = point_n;
       q->th = _th;
       q->zero = false;
       q->cluster = (int*)malloc(sizeof(int)*point_n);
       memset(q->cluster, 0, sizeof(int)*point_n);
       q->next_th = data;
       data = q;
       return;
}
```

Per ogni nuova soglia di binarizzazione, viene creata una pagina nella struttura dati *tab*; ogni pagina tiene conto del numero di punti presenti nel dataset, di un puntatore ad una vettore di interi che rappresenta i cluster assegnati ad ogni punto, della soglia di binarizzazione alla quale si riferisce e di un valore booleano che indica se è presente o meno almeno un assegnazione di un punto a zero (quindi a nessun cluster).

La funzione che assegna i cluster ai punti opera nel seguente modo:

- Per ogni punto del dataset:
  - Estrae le sue coordinate e calcola l'indice della griglia che è più vicino al punto;
  - Assegna il label del punto della griglia a quel punto;

L'implementazione del codice è riportata di seguito:

```
void assignPointsToCluster(double* cf, int dim, int n, double* points, int patterns, tab*
data){
double res = 0;
res = (double)1 / (double)(n - 1);      //risoluzione della griglia dati n punti
int index = 0;
int i = 0, j = 0;
int temp = 0;
data->zero = false;
for (i = 0; i < patterns; i++){
       //per ogni punto
       for (j = 0; j < dim; j++){
       //per ogni coordinata calcola la sua proiezione nella griglia
              temp = (int)round((points[(i*dim) + j] / res));
              index = index + (int)(pow((double)n, (double)j)*temp);
       }
       if (index < (int)pow((double)n, (double)dim)){
              data->cluster[i] = (int)cf[index];
              if (data->cluster[i] == 0) data->zero = true; //segna se sono stati trovati
degli zeri
       }
       index = 0;
}
}
```

Produzione dei file di uscita in Visual Studio: CLS_th_oxx.txt e Res_th_oxx.txt

Tutto quello detto fino ad ora, viene incapsulato in un ciclo *for* che fa variare il valore della soglia sulla quale le funzioni operano. Di seguito viene riportato il codice implementato[4] con commenti a posteriori:

```
int i = 0;
int j = 0;
int k = 0;
int end_index = (int)pow((double)n, (double)dim);    //variabile utile a non far eseguire
questo calcolo su device (vedi binarize())
double* dev_bf;
cudaMalloc(&dev_bf, sizeof(double)*(int)pow((double)n, (double)dim));
char name_file[15];
memset(name_file, 0, sizeof(name_file));
char tempStr[sizeof(int)*10+1];
memset(tempStr, 0, sizeof(name_file));
double th = 0.99;
tab* data = new tab;
data = NULL;
```

Questa parte prevede di istanziare e inizializzare le variabili necessarie al corretto funzionamento.

```
for (i = 99; i > 0; i--){
       //ciclo che esegue il cambiamento delle soglia della binarize
       cudaMemset(dev_bf, 0, sizeof(double)*(int)pow((double)n, (double)dim));
       //resetta il valore dei dati presenti nella variabile di appoggio dev_bf
       cudaMemcpy(dev_bf, dev_mf, sizeof(double)*(int)pow((double)n, (double)dim),
cudaMemcpyDeviceToDevice);
       //copia il valore dell mf dentro la dev_bf
       binarize << <(int)pow((double)n, (double)dim - 1), n >> > (dev_bf, dim, n, th,
end_index);
```

[4] Sono state omesse le parti per la verifica degli errori CUDA per una più chiara leggibilità.

```
        cudaDeviceSynchronize();
        cudaMemcpy(mf, dev_bf, sizeof(double)*(int)pow((double)n, (double)dim),
cudaMemcpyDeviceToHost);
        //sposta i dati da memoria device a host
        findCluster(mf, dim, n);
        new_tab(patterns, th, data);
        assignPointsToCluster(mf, dim, n, points, patterns, data);
                //composizione del nome file per i cluster trovati
        memset(name_file, 0, sizeof(name_file));
        memset(tempStr, 0, sizeof(tempStr));
        if (i < 10){
                strncpy(name_file, "Res_th_00", 9);
                itoa(i, tempStr, 10);
                strncpy(&name_file[9], tempStr, 1);
                strcpy(&name_file[10], ".txt");
        }
        else{
                strncpy(name_file, "Res_th_0", 8);
                itoa(i, tempStr, 10);
                strncpy(&name_file[8], tempStr, 2);
                strcpy(&name_file[10], ".txt");
        }
        fp = fopen(name_file, "w");
        for (k = 0; k<(int)pow((double)n, (double)dim - 1); k++){
                for (j = 0; j<n; j++)
                        fprintf(fp, "%lf\t", mf[n*k + j]);
                fprintf(fp, "\n");
        }
        fprintf(fp, "\n\n");
        fclose(fp);
        memset(name_file, 0, sizeof(name_file));
        memset(tempStr, 0, sizeof(tempStr));
        if (i < 10){
                strncpy(name_file, "CLS_th_00", 9);
                itoa(i, tempStr, 10);
                strncpy(&name_file[9], tempStr, 1);
                strcpy(&name_file[10], ".txt");
        }
        else{
                strncpy(name_file, "CLS_th_0", 8);
                itoa(i, tempStr, 10);
                strncpy(&name_file[8], tempStr, 2);
                strcpy(&name_file[10], ".txt");
        }
        f3 = fopen(name_file, "w");
                //scrive il file per poter essere aperto in matlab
        for (k = 0; k< data->point_numbers; k++) fprintf(f3, "%d\t", data->cluster[k]);
        fclose(f3);
        th = th - 0.01;
        }
```

Il ciclo *for* chiama in sequenza le funzioni di cui si è parlato nei precedenti paragrafi, preoccupandosi di spostare le strutture dati tra memoria GPU e CPU e salvare i file di assegnazione dei punti ai cluster (CLS_th_oxx.txt) e l'immagine contenente i cluster stessi (Res_th_oxx.txt).

```
        //scrittura log file
diff = clock() - start;
int msec = diff * 1000 / CLOCKS_PER_SEC;
FILE* f2 = fopen("log.txt", "a");
fprintf(f2, "------------\nNEW
DATA\ndim:\t\t%d\ngamma:\t\t%lf\np_dataset:\t%d\nRes:\t\t%d\nTotal Time:\t%d ms\n",
                dim, gamma, patterns, n, msec);
fclose(f2);
printf("Tempo di esecuzione:\t%d ms\n", msec);
        //liberazione memoria occupata
cudaFree(dev_distances);
cudaFree(dev_mf);
cudaFree(dev_bf);
free(points);
```

```
free(mf);
free(data);
return 0;
```

Mentre a seguito del ciclo *for* di variazione delle soglie, viene salvato un file di log sullo storico delle analisi eseguite e liberate le strutture dati allocate in GPU e CPU.

DESCRIZIONE DEI BLOCCHI COSTITUTIVI – Matlab

Per poter gestire la creazione del file 'albero binario' da dare in ingresso alla funzione dendrogram di Matlab, sono state create delle funzioni utili che di seguito verranno elencate e descritte.

importresfile.m

Questa funzione permette di importare come struttura dati all'interno del programma Matlab il file Res_th_oxx.txt passando come parametro unicamente un indice che descrive quale soglia si vuole importare. Di seguito è riportata la sua implementazione:

```
%%data=importresfile importa i dati dei file CLS
%%parametri di input:
%%index: l'index del file Res_th_xxx.txt da importare
%%Ritorno:
%%data: dati della griglia

function data=importresfile(index)
if index>0 && index<=99,
if index<10,
    temp1='Res_th_00';
else
    temp1='Res_th_0';
end;
temp2=num2str(index);
temp3='.txt';
file_to_import = [temp1 temp2 temp3];   %queste righe preparno la stringa per leggere i
dati CLS correnti
data = importdata(file_to_import);
else
    error('Index non è nel range consentito!');
end;
end
```

Una volta importato su una variabile di appoggio, sarà possibile graficare il risultato della clusterizzazione[5] tramite il comando:

```
data=importresfile(40);
pcolor(data);
```

importclsfile.m

La funzione, alla stregua della precedente, permette di importare come dati utili i valori contenuti nel file CLS_th_oxx.txt, passandogli unicamente l'indice di nostro interesse; l'implementazione è la seguente:

[5] Nel caso 2D soltanto.

```
%%[cls_data n_clus]=importclsfile importa i dati dei file CLS
%%parametri di input:
%%index: l'index del file CLS da importare
%%Ritorno:
%%cls_data: un vettore di dati rappresentante le assegnazioni dei cluster
%%ai punti
%%n_clus: il numero di cluster presenti

function [cls_data n_clus]=importclsfile(index)
if index>0 && index<=99,
if index<10,
    temp1='CLS_th_00';
else
    temp1='CLS_th_0';
end;
temp2=num2str(index);
temp3='.txt';
file_to_import = [temp1 temp2 temp3];   %queste righe preparno la stringa per leggere i
dati CLS correnti
cls_data = importdata(file_to_import);
n_clus=length(unique(cls_data));       %estrae il numero di cluster dal file
else
    error('Index non è nel range consentito!');
end;
end
```

In questo caso i valor di ritorno sono due: il primo (cls_data) rappresenta effettivamente il vettore contenente i cluster assegnati; *n_clus*, invece, indica quanti cluster totali sono presenti nell'assegnazione.

clusinfo.m

Questa funzione permette di ottenere alcune informazioni inerenti il cluster presente nella struttura dati *cls_data* il cui inizio è identificato dal parametro *index* di passaggio. Si riporta e commenta il codice:

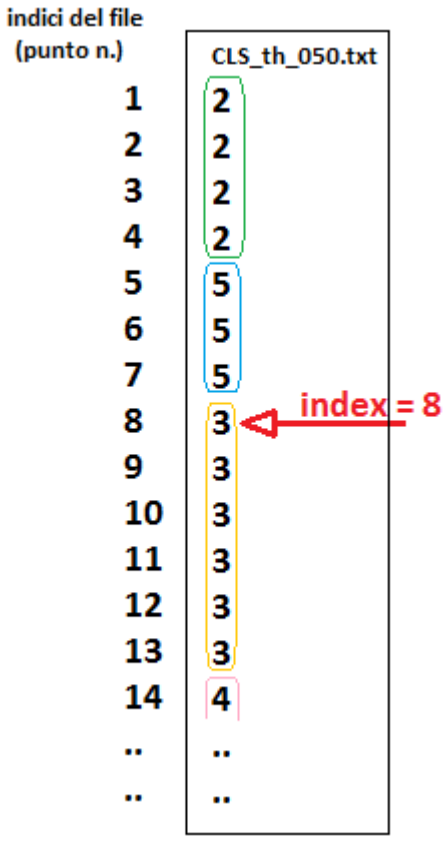
```
%%[start end_index clus_length]=clusinfo(cls_data, index) ritorna l'index di inizio, fine
e lunghezza
%%del cluster che parte dalla posizione index del vettore cls_data
%%Parametri:
%%cls_data: vettore cls
%%index: indice da cui considerare il cluster
%%Ritorno:
%%start: indice di inizio
%%end_index: indice di fine
%%length: lunghezza del cluster

function [start end_index clus_length]=clusinfo(cls_data, index)
start=index;
end_index=index;
clus_length=1;
pattern=length(cls_data);
if (index+1)<=pattern && cls_data(index)==cls_data(index+1),
    %scansiona l'intero vettore CLS e confronta due punti adiacenti
clus_length=2;
index=index+1;
while((index+1)<=pattern && cls_data(index)==cls_data(index+1)),
  index=index+1;
  clus_length=clus_length+1;
end;
end_index=index;
end;
end
```

I valori che la funzione ritorna sono tre:

- L'indice di inizio del cluster (*start*);
- L'indice di fine del cluster (*end_index*);
- La lunghezza del cluster (*clus_length*);

Per meglio comprendere il funzionamento della funzione si può far riferimento alla seguente immagine:



In questo caso, ad esempio, è stata chiamata la funzione:

```
r50=imporclsfile(50);
[start end_index clus_length]=clusinfo(r50, 8);
```

E i valori contenuti nelle tre variabili di uscita saranno:

```
start = 8;
end_index = 13;
clus_length = 6;
```

clus.m

Questa funzione permette di richiamare e configurare l'applicazione generata dal programma Visual Studio in base ai parametri di ingresso passati; il codice è il seguente:

```
%clus visualizza i risultati della clusterizzazione del file di
%ingresso per 4 diversi valori di soglia (0.2, 0.4, 0.6, 0.8)
%Params:
%dim:        dimensions
%gamma:      gamma
%res:        grid resolution (number of points between 0 and 1)
%points:     number of points
%In:         Dataset as file text name
%enable_fig: 1 or 0 per plottare(enable_fig=1) o meno le figure

function z = clus(dim,gamma,res,In,enable_fig)
    X=importdata(In);
    [R,C] = size(X);
    z=R;
    cmd=['Pervasive_Systems_project.exe ',num2str(1),' ', num2str(dim),' ',
num2str(gamma),' ', num2str(R),' ', num2str(res),' ', In];
    system(cmd);
    if enable_fig == 1,
        %stampa solo se viene richiesto
    scrsz = get(0,'ScreenSize');    %prende la grandezza dello schermo
    d20=importdata('Res_th_020.txt');
    d40=importdata('Res_th_040.txt');
```

```
    d60=importdata('Res_th_060.txt');
    d80=importdata('Res_th_080.txt');
    figure('OuterPosition',[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2],'Name','Result
th=0.2','NumberTitle','off'); %posiziona la finestra in alto a sx
    pcolor(d20);
     figure('OuterPosition',[scrsz(3)/2 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2],'Name','Result
th=0.4','NumberTitle','off'); %posiziona la finestra in alto a dx
    pcolor(d40);
     figure('OuterPosition',[scrsz(2)/2 1 scrsz(3)/2 scrsz(4)/2],'Name','Result
th=0.6','NumberTitle','off'); %posiziona la finestra in basso a sx
    pcolor(d60);
     figure('OuterPosition',[scrsz(3)/2 1 scrsz(3)/2 scrsz(4)/2],'Name','Result
th=0.8','NumberTitle','off'); %posiziona la finestra in basso  dx
    pcolor(d80);
    figure('Name', 'Cluster assegnation');
    c20=importdata('CLS_th_020.txt');
    c40=importdata('CLS_th_040.txt');
    c60=importdata('CLS_th_060.txt');
    c80=importdata('CLS_th_080.txt');
    subplot(4,1,1);
    plot(c20);
    subplot(4,1,2);
    plot(c40);
    subplot(4,1,3);
    plot(c60);
    subplot(4,1,4);
    plot(c80);
    end
```

Oltre ai parametri necessari alla configurazione dell'applicativo Visual Studio, c'è un ultimo parametro, *enable_fig* che, se uguale a '1', permette di plottare a schermo i risultati di clusterizzazione per 4 diverse soglie di binarizzazione: 0.2, 0.4, 0.6, 0.8. Questa operazione era utile più che altro per una fase iniziale di debug, ma è comunque rimasta implementata.

fdbdcls.m

Questa funzione, il cui acronimo vuole significare ***F**ind **D**istance **B**etween **D**ifferent **CLS** file*, permette di trovare, dato un indice di binarizzazione, *index*, quale sia il primo precedente indice di binarizzazione che presenta un numero di cluster minore di quello attuale. Questa informazione è molto utile, come vedremo nel seguito, per la ricostruzione del file albero binario e dunque del dendrogramma. Il codice è il seguente:

```
%%[diff distance curr_cls_index prev_cls_index curr_cls prev_cls]=fdbdcls(index) find
distance between different cls file
%%trova la distanza e il numero di cluster presenti fra due diversi file cls
%%parametri di input:
%%index:            indice del file cls da cui partire a ritroso
%%Ritorno:
%%diff:             è la distanza in numero di cluster trovata
%%distance:         è la distanza in numero di soglie th tra i due file trovati
%%curr_cls_index:   è l'indice del file cls corrente
%%prev_cls_index:   è l'indice del file cls che ha un numero di cluster
                    presenti inferiore a quello corrente
%%curr_cls:         è il vettore cls corrente
%%prev_cls:         è il vettore cls precedente che presenta un numero di cluster
                    minore

function [diff distance curr_cls_index prev_cls_index curr_cls prev_cls]=fdbdcls(index)
[curr_cls, curr_n_clus]=importclsfile(index);
curr_cls_index=index;
k=index-1;
if (k)>0,
```

```
 [prev_cls, prev_n_clus] = importclsfile(k);
  while prev_n_clus == curr_n_clus,
     %%sposta l'indice prev_CLS fintanto ce non trova due valori
     %%differenti di numero di cluster assegnati
    k=k-1;
    if k>0,
       %%tutto ok, continuiamo a scansionare i file CLS_th_..
      [prev_cls, prev_n_clus]=importclsfile(k);
    else
       %%abbiamo raggiunto l'inizio dei file CLS
       diff=0;
       prev_cls_index=index;
       distance=0;
       return;
    end;
  end;
 diff = curr_n_clus - prev_n_clus;
 prev_cls_index=k;
 distance=curr_cls_index-prev_cls_index;
else
  disp('Raggiunto inizio file22');
  diff=0;
  prev_cls_index=index;
  distance=0;
  return;
end;
end
```

La funzione ritorna inoltre altri parametri interessanti per la costruzione del dendrogramma:

- La differenza nel numero di cluster trovati (*diff*);
- La distanza, in numero di soglie di binarizzazione, tra i due indici trovati (*distance*);
- L'indice passato come corrente (*curr_cls_index*);
- L'indice del file CLS che presenta un numero di cluster minore del corrente (*prev_cls_index*);
- L'immagine corrente (*curr_cls*);
- L'immagine relativa all'indice trovato (*prev_cls*);

findClusIndex.m

Prima di poter descrivere questa funzione, si deve introdurre la struttura dati 'albero binario'.

Dati M punti, un file 'albero binario' è una matrice (M – 1) x 3 dove le prime due colonne identificano i cluster da unire nel dendrogramma, mentre la terza colonna identifica la distanza alla quale devono essere uniti[6]. Ogni nuova riga della matrice identifica un nuovo livello, il cui identificativo sarà dato dalla somma del numero di riga e del numero totale di punti presenti.

La funzione che opera la costruzione di questa struttura dati, analizeClusterData.m, dovrà quindi utilizzare una matrice (M – 1) x 3[7].

Si viene ora alla spiegazione della findClusIndex.m: durante lo svolgimento dell'algoritmo principale, si ha la necessità di capire quale sia l'ultimo indice di livello utile a cui il punto che dobbiamo inserire nel dendrogramma fa riferimento. Si sottolinea infatti che la matrice *data_matrix* è una matrice che in partenza

[6] Nel nostro caso questo valore sarà banalmente 1-th.

[7] In realtà, la funzione utilizzerà una matrice (M – 1) x 4 che poi sarà ridotta nuovamente a (M – 1) x 3. Il valore contenuto nella quarta colonna è il valore corrente del livello a cui la riga fa riferimento. La quarta colonna è stata introdotta unicamente per semplicità e velocità dell'algoritmo.

è costituita da soli zeri e viene riempita poco alla volta durante lo svolgimento dell'algoritmo. Si riporta il codice della findClusIndex:

```
%%cluster_index=findClusIndex(data_matrix, point_number) ritorna l'indice
%%utile del cluster relativo al punto richiesto salvato nella data_matrix,
%%matrice (M-1)x4, con M numero di punti del dataset, le cui prime 3 righe
%%sono organizzate secondo il formalismo dei dati in ingresso della
%%funzione dendrogram. La data_matrix si considera ancora incompleta, e
%%deve essere creta come tutti zeri, le cui righe vengono riempite man mano.
%%la quarta colonna contiene gli indici con cui viene identificato il
%%cluster della riga.
%%parametri di ingresso:
%%data_matrix:      è la matrice su cui vengono letti i dati
%%point_number:     è l'indice del punto da cui si vuole risalire l'ultimo
%%                  cluster di appartenenza

function cluster_index=findClusIndex(data_matrix, point_number)
[R C] = size(data_matrix);
cluster_index=point_number;
for i=1:R,
       %scansiona tutte le righe
    temp_row=data_matrix(i,:);
    if temp_row==zeros(1,4),
        %le restanti righe non possono che essere tutte zero
        return;
    end;
    if (temp_row(1)==cluster_index) || (temp_row(2)==cluster_index),
        cluster_index=temp_row(4);
    end;
end;
end
```

analizeClusterData.m

Questa è l'unica funzione che si dovrà utilizzare per il funzionamento del progetto. Si preoccuperà di richiamare l'applicativo Visual Studio e tutte le precedenti funzioni Matlab viste, per andare a costruire il dendrogramma e plottare quello che rappresenta l'immagine più realistica del raggruppamento di cluster.

Il funzionamento dell'algoritmo è diviso in 3 fasi differenti:

- **Fase 1**: ricerca del primo file CLS utile.

In questa fase l'algoritmo scandisce tutti i file cls partendo dall'indice 99 a scendere, per trovare una delle due seguenti condizioni:

  - Numero di cluster presenti eguaglia il numero di pattern (Condizione di partenza ottimale[8]);
  - Numero di cluster presenti maggiore di tutti gli altri file CLS, ma che non contiene al suo interno 0, quindi tutti i punti sono assegnati ad un cluster.

- **Fase 2**: unione dei primi cluster del dendrogramma.

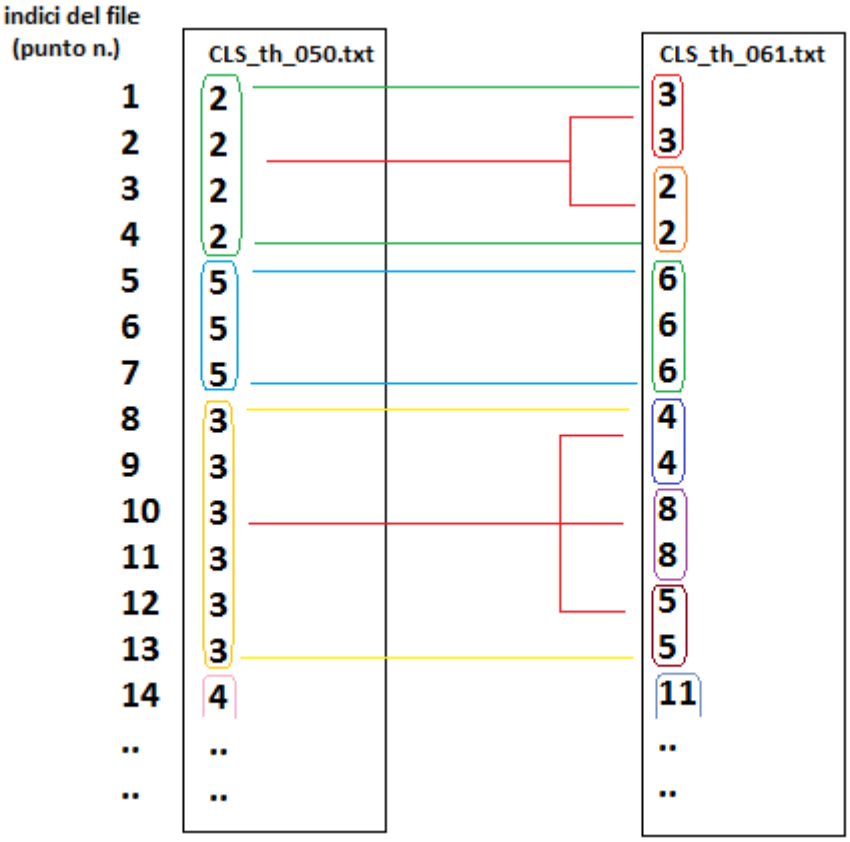
Questa fase serve per inserire le prime righe della matrice *data_matrix* qualora la condizione ottimale non sia soddisfatta: in questo caso infatti non si potrà procedere con il normale svolgimento dell'algoritmo ma si dovranno creare i gruppi di punti appartenenti allo stesso cluster presenti già in avvio.

[8] Infatti partiamo con una condizione in cui ogni punto rappresenta un cluster a se stante.

- **Fase 3**: creazione del dendrogramma.

Qui si scorrono tutti gli indici dei file CLS che presentano differenze nel numero di cluster contenuti:

- Una volta trovati i due file **CLS presente** e **CLS precedente,** si scorre all'interno del file precedente:
    - Per ogni cluster si rintraccia inizio e fine, si confronta con il corrispettivo file CLS presente e, nel caso gli indici di inizio e fine del precedente contengono più cluster del presente, allora vengono inserite delle righe nella matrice data_matrix per aggiungere questa unione.



Ogni volta che deve essere inserita una nuova riga, si deve rintracciare l'ultimo livello della *data_matrix* a cui il punto da inserire fa riferimento, e quindi viene utilizzata la funzione findClusIndex().

Durante la scansione, si tiene conto di quale sia stata la distanza maggiore riscontrata tra due diversi file CLS, e sarà proprio questa distanza che individuerà i due indici CLS che saranno utilizzati per il plot del grafico rappresentativo.

La scansione dei file termina o per il raggiungimento del file CLS_th_001.txt o ugualmente perché non si trovano differenze nel numero di cluster presenti tra CLS presente e CLS precedente. Il corpo della funzione è il seguente:

```
%%[dataResult cls_index_found num_cluster]=analizeClusterData(dim,gamma,res,In)
%Params:
%dim:        dimensions
%gamma:      gamma
%res:        grid resolution (number of points between 0 and 1)
%points:     number of points
%In:         Dataset
%RETURN:
%dataResult:     i dati dell'immagine scelta
%cls_index:      l'indice della soglia relativa all'immagine scelta
%num_cluster:    numero di cluster presenti nell'immagine
```

```
function [dataResult cls_index_found num_cluster]=analizeClusterData(dim,gamma,res,In)
disp('Aspettando l elaborazione CUDA/C++...');
n_pattern = clus(dim,gamma,res,In,0);
disp('Computazione eseguita.');
data_matrix=zeros(n_pattern-1,4);
%inizializzazione dello script, ricerca del primo file CLS utile
cls_index=99;
max_cluster_found=0;
curr_n_clus=0;
max_cls_index=0;
%INIZIO FASE 1
while cls_index>0,
    %cicla finchè non si raggiunge la condizione di inizio
[curr_CLS, curr_n_clus]=importclsfile(cls_index);
if curr_n_clus==n_pattern,
    %condizione iniziale ottima
    max_cls_index=cls_index;
    disp('Condizione di partenza ottimale');
    initial_iterations=0;
    break;
elseif curr_n_clus<n_pattern && min(curr_CLS)>1,
    if curr_n_clus>max_cluster_found,
        max_cluster_found=curr_n_clus;
        max_cls_index=cls_index;
        initial_iterations=n_pattern-max_cluster_found;
    end;
end;
cls_index=cls_index-1;
end;
cls_index=max_cls_index;
%%adesso abbiamo trovato il primo file CLS che contiene un numero di
%%cluster pari al numero di pattern
rows=1;
diff=1; %%differennza di numero di cluster trovati tra curr e prev
max_difference_found=0; %%massima lontanaza tra due indici con lo stesso numero di
cluster
    %dobbiamo creare 'initial_iteration' cluster
%INIZIO FASE 2
    point3_to_insert=0.0005;
    initial_cluster=importclsfile(max_cls_index);
    a=1;
    while a<=n_pattern,
       [start_clus end_clus clus_length]=clusinfo(initial_cluster, a);
       if clus_length==2,
             point1_to_insert=start_clus;
             point2_to_insert=end_clus;
             point4_to_insert=rows+n_pattern;
             data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
             rows=rows+1;
        elseif clus_length>2,
             point1_to_insert=a;
             point2_to_insert=a+1;
             point4_to_insert=rows+n_pattern;
             data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
             rows=rows+1;
             a=a+2;
            while a<=end_clus,
             point1_to_insert=point2_to_insert;
             point2_to_insert=a;
             point4_to_insert=rows+n_pattern;
             data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
             rows=rows+1;
             a=a+1;
            end;
       end;
        a=end_clus+1;
    end;
```

```
    h = waitbar(0.0001,'Sto creando il dendrogramma..','Name','Solo un momento..');
%INIZIO FASE 3
while cls_index>1 && diff>0,
    %%questa parte serve a gestire il caso in cui non si riscontra la
    %%condizione ottimale di partenza, dunque si devono creare i cluster di
    %%inizio
    if rows>n_pattern,
        error('rows è cresciuto troppo');
    end;
 [diff distance curr_cls_index prev_cls_index curr_CLS prev_CLS]=fdbdcls(cls_index);
if distance>max_difference_found,
     %tiene conto del cluster che si mantiene più a lungo
     max_difference_found=distance;
     cls_to_print=prev_cls_index+((curr_cls_index-prev_cls_index)/2);
 end;
  j=1;                %indice per scansionare internamente il file CLS
  for z=1:diff,
      %per ogni nuovo cluster trovato
   for fff=1:n_pattern,
   [prev_start_clus prev_end_clus prev_clus_length]=clusinfo(prev_CLS, j);
      %%adesso abbiamo in prev_start_clus, prev_end_clus e
      %%prev_clus_length l'indice di inizio e fine del cluster che stiamo
      %%osservando del vettore CLS e la lunghezza di questo cluster
    sub_clus_present=0; %%aiuta a capire quanti sotto cluster sono presenti in curr_cls
    j=prev_start_clus;        %pone l'indice di inizio in curr_CLS pari a j
    while j<=prev_end_clus,
    [curr_start_clus curr_end_clus curr_clus_length]=clusinfo(curr_CLS, j);
    if (curr_end_clus <= prev_end_clus),
            %%il cluster corrente in curr_CLS deve essere unito con quello
            %%padre in prev_CLS
        %point_to_insert=curr_end_clus;    %è il punto da inserire
        sub_clus_present=sub_clus_present+1;
        if bitget(sub_clus_present,1),
            %caso dispari
            if sub_clus_present>2
                %in questo caso si devono unire più cluster, dunque le
                %righe della matrice che dobbiamo costruire saranno diverse
                point1_to_insert=point2_to_insert;  %è quello precedente in posizione
"point2"
                point2_to_insert=findClusIndex(data_matrix, curr_end_clus);
                point3_to_insert=1-(curr_cls_index/100);
                point4_to_insert=rows+n_pattern;
                data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
                rows=rows+1;
            else
                point1_to_insert=findClusIndex(data_matrix, curr_end_clus);
            end;

        else
            %caso pari
            point3_to_insert=1-(curr_cls_index/100);
            point4_to_insert=rows+n_pattern;
            if sub_clus_present>2
                point1_to_insert=point2_to_insert;  %è quello precedente in posizione
"point2"
                point2_to_insert=findClusIndex(data_matrix, curr_end_clus);
                data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
                rows=rows+1;
            else
                point2_to_insert=findClusIndex(data_matrix, curr_end_clus);
                data_matrix(rows,:)=[point1_to_insert point2_to_insert point3_to_insert
point4_to_insert];
                rows=rows+1;
            end;
        end;
     else
          %%non dovrebbe verificarsi questa situazione
          error('curr_end_clus>prev_end_clus..non può essere!');
```

```
     end;
     j=curr_end_clus+1;   %continua finchè non trova la fine del cluster
    end;
   end;

  end;
    cls_index=prev_cls_index;
    waitbar((1-(curr_cls_index/100)),h);
end;
waitbar(1,h);

app=data_matrix([1:(n_pattern-1)],[1 2 3]);
dendrogram(app, n_pattern);
xlabel('Punti del dataset');
ylabel('1 - <<Soglie di binarizzazione>>');
title('Dendrogramma');
if dim<=2,
    %stampa solo se la dimensione è 2 altrimenti dimensioni superiori non
    %hanno modo di poter essere stampate
figure;
cls_to_print=round(cls_to_print);
grid_data=importresfile(cls_to_print);
pcolor(grid_data);
xlabel('Asse x della griglia');
ylabel('Asse y della griglia');
str = sprintf('Miglior Cluster %d', cls_to_print);
title(str);
end;
delete(h);  %cancella la waitbar
dataResult=grid_data;
cls_index_found=cls_to_print;
num_cluster=length(unique(importclsfile(cls_to_print)));
end
```

Il grafico rappresentativo sarà quello situato a metà strada tra i due indici CLS presente e precedente.

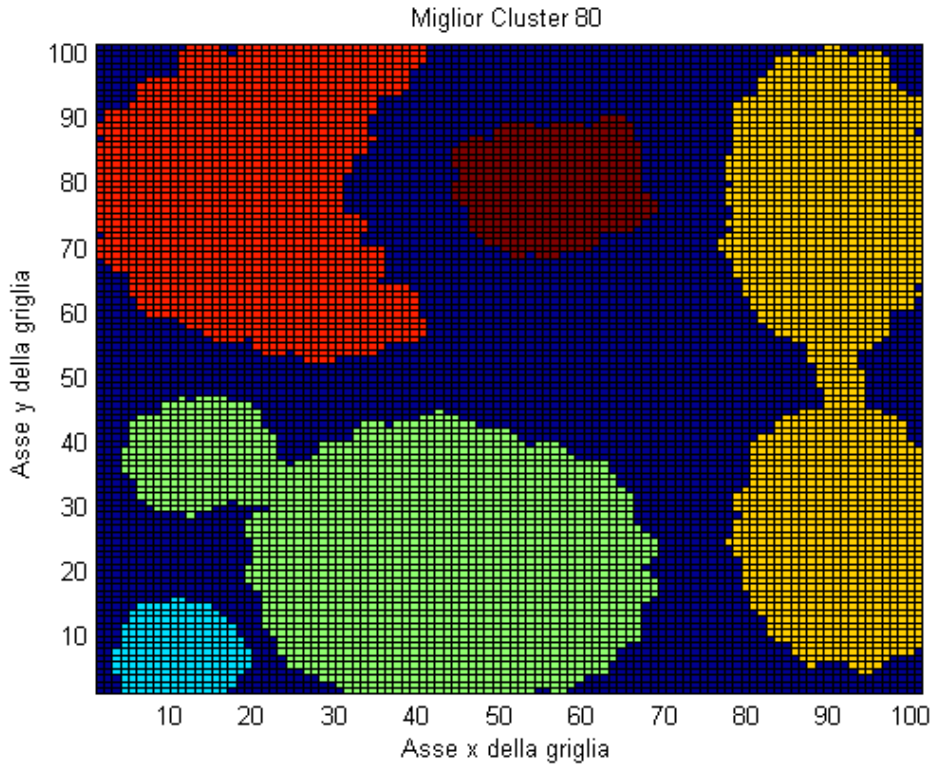
RISULTATI RISPETTO I DATASET FORNITI

Si elencano di seguito i dataset a disposizione:

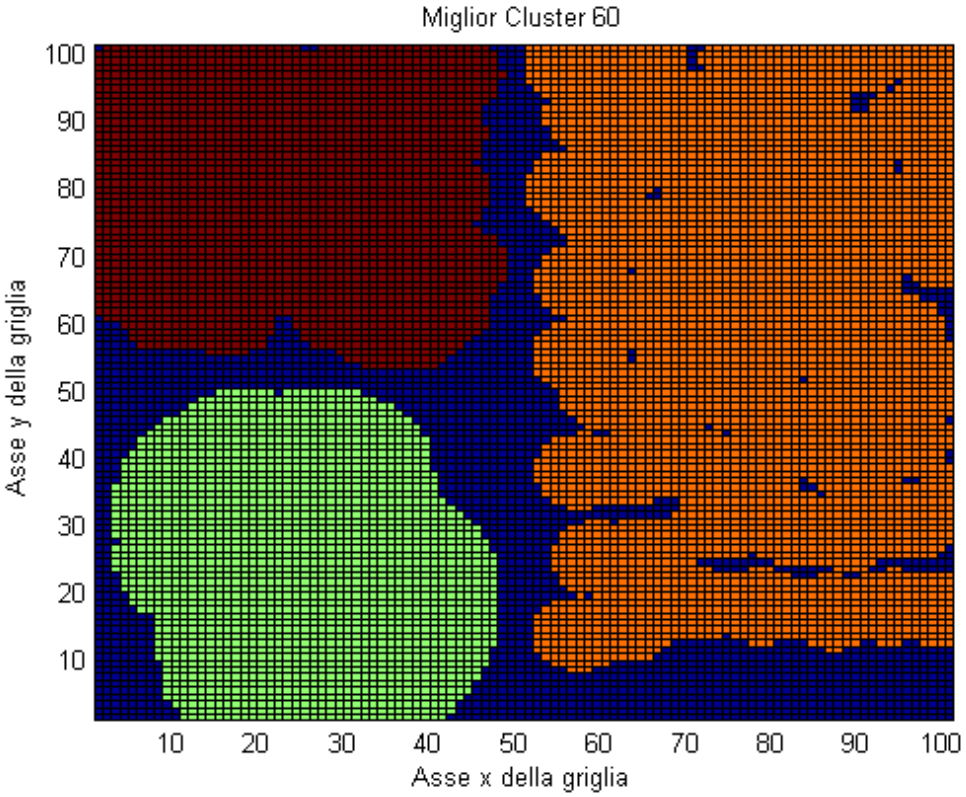
Nome del dataset	**Numero di punti**	**Dimensioni**	**Numero di cluster**
aggregation	788	2	7
compound	399	2	6
d31	3100	2	31
flame	240	2	2
jain	373	2	2
pathbased	300	2	3
r15	600	2	15
spiral	312	2	3
lsun	400	2	3

Adesso si passa a elencare i risultati ottenuti per ognuno dei dataset in tabella, eseguiti con una risoluzione di 101 punti per asse e un valore di gamma pari a 10.

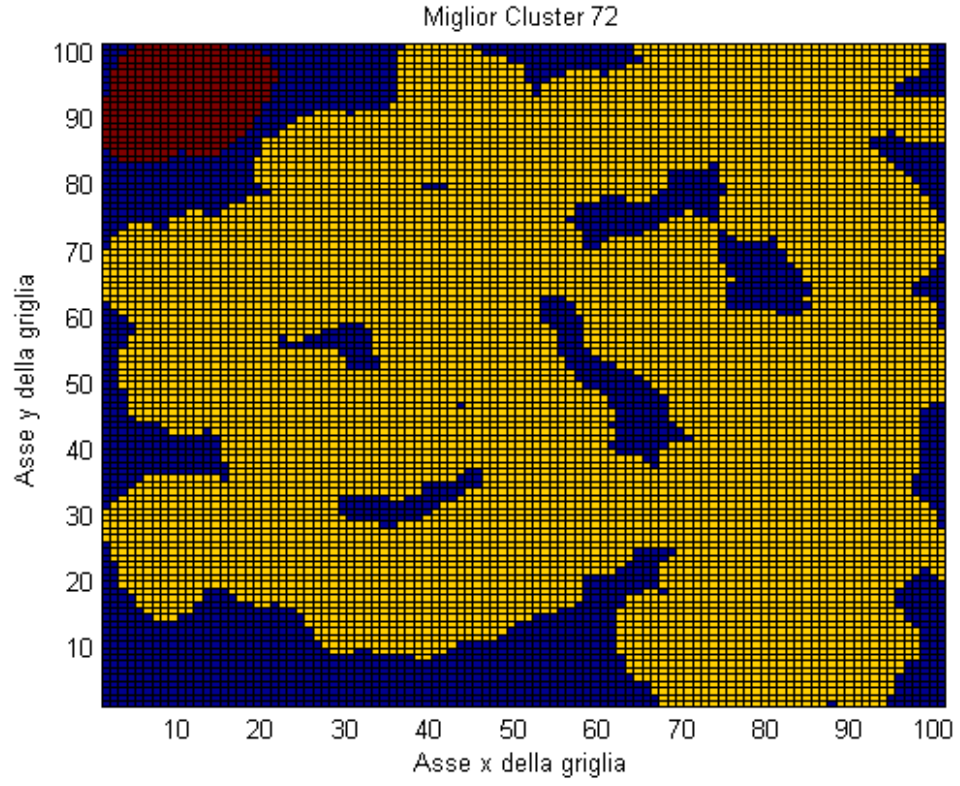
AGGREGATION



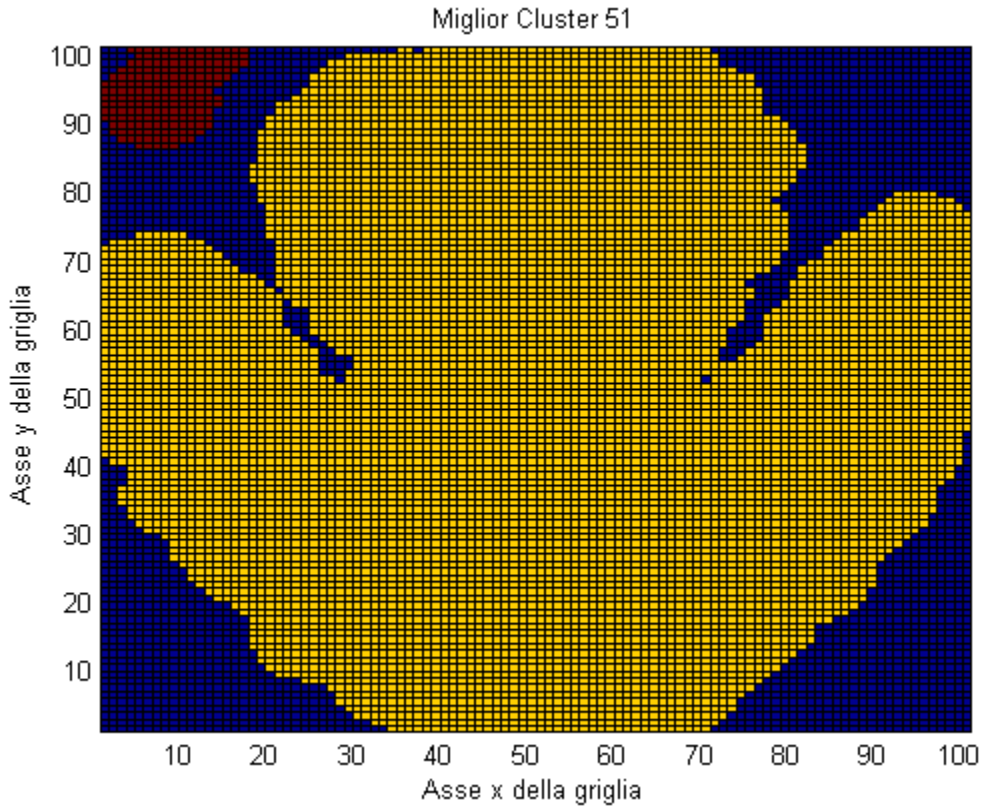
COMPOUND



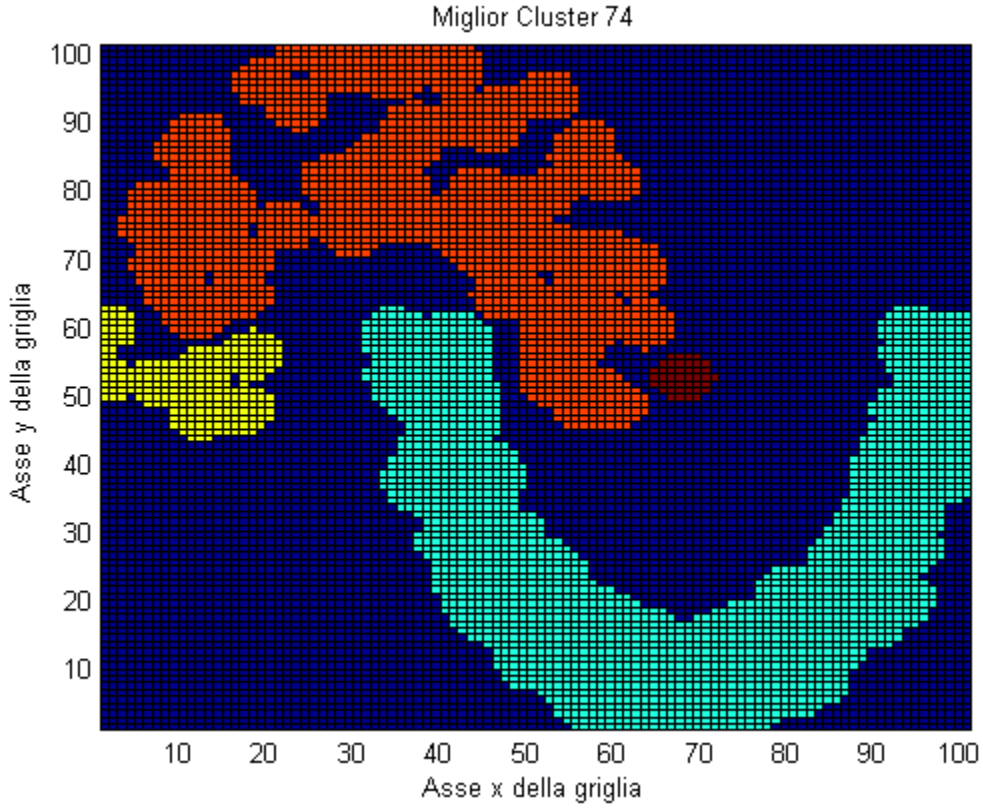
D31



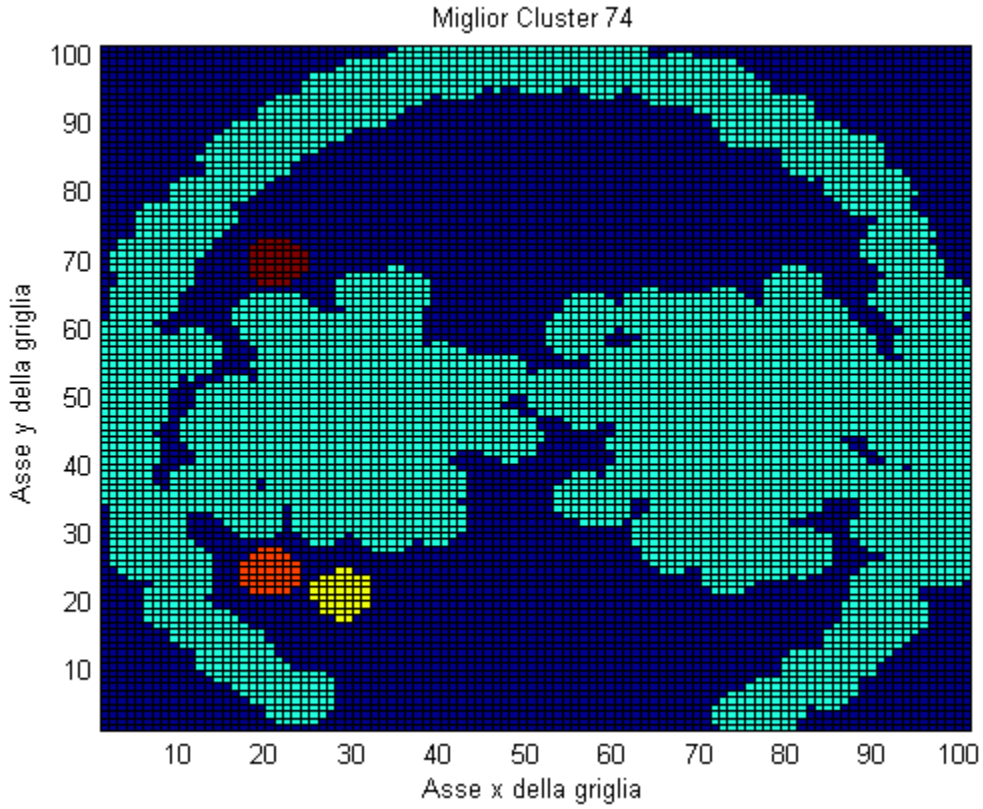
FLAME


Miglior Cluster 51
Asse y della griglia
Asse x della griglia
10
20
30
40
50
60
70
80
90
100

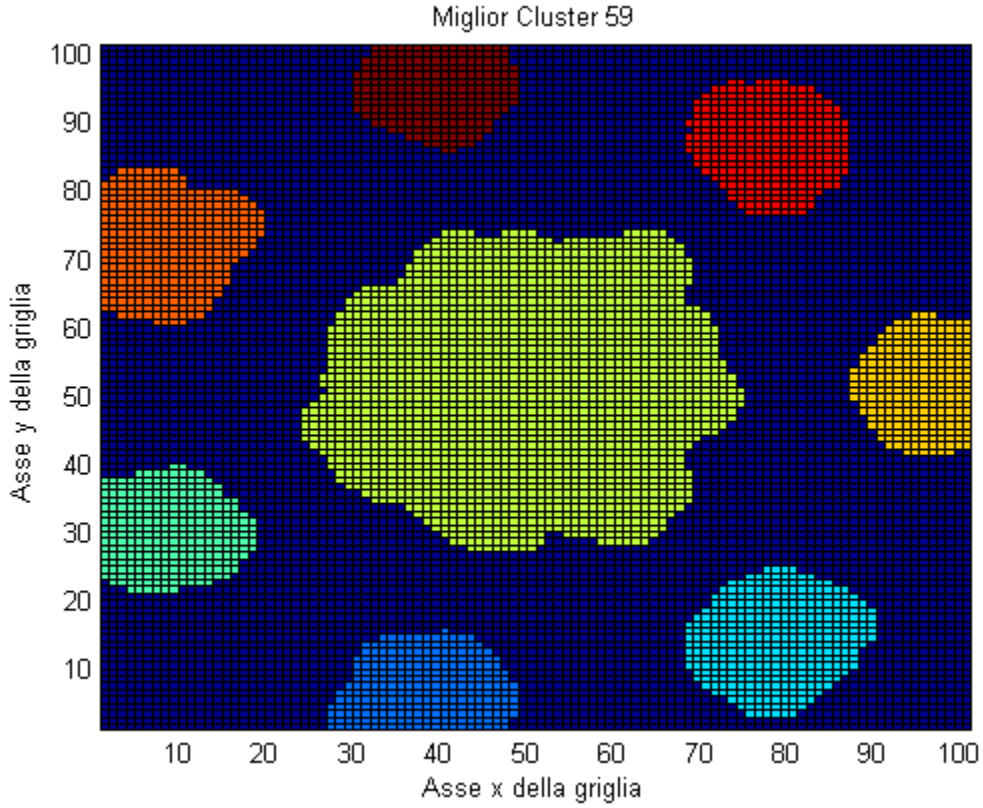
JAIN


Miglior Cluster 74
Asse y della griglia
Asse x della griglia
10
20
30
40
50
60
70
80
90
100

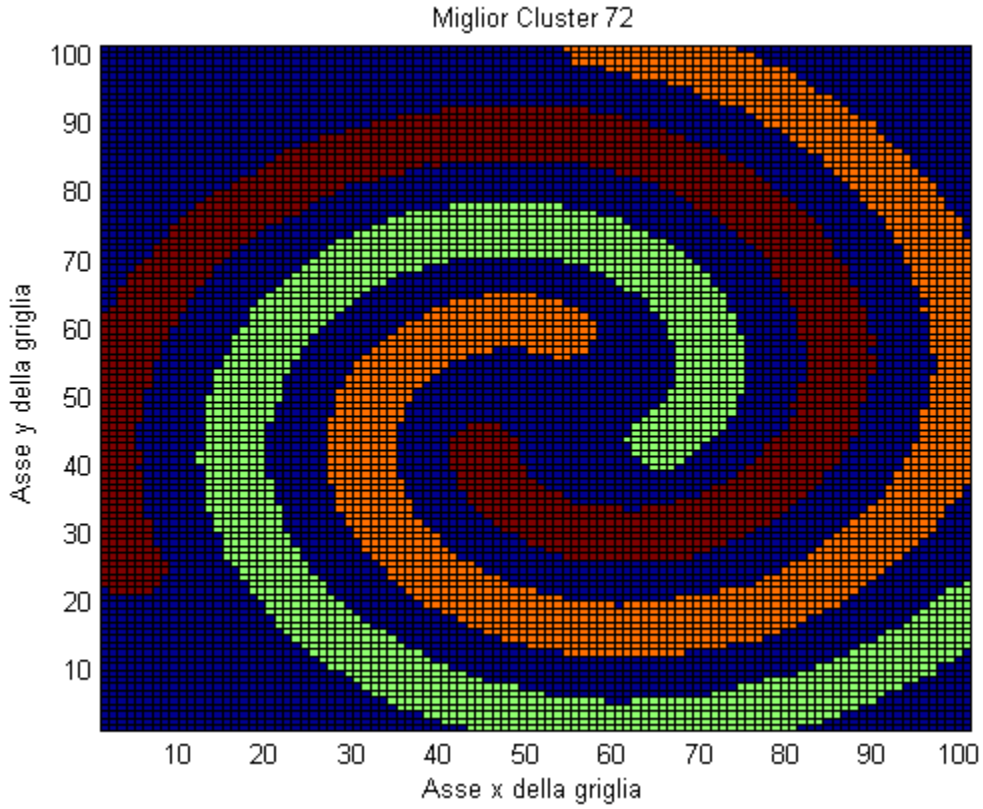
PATHBASED



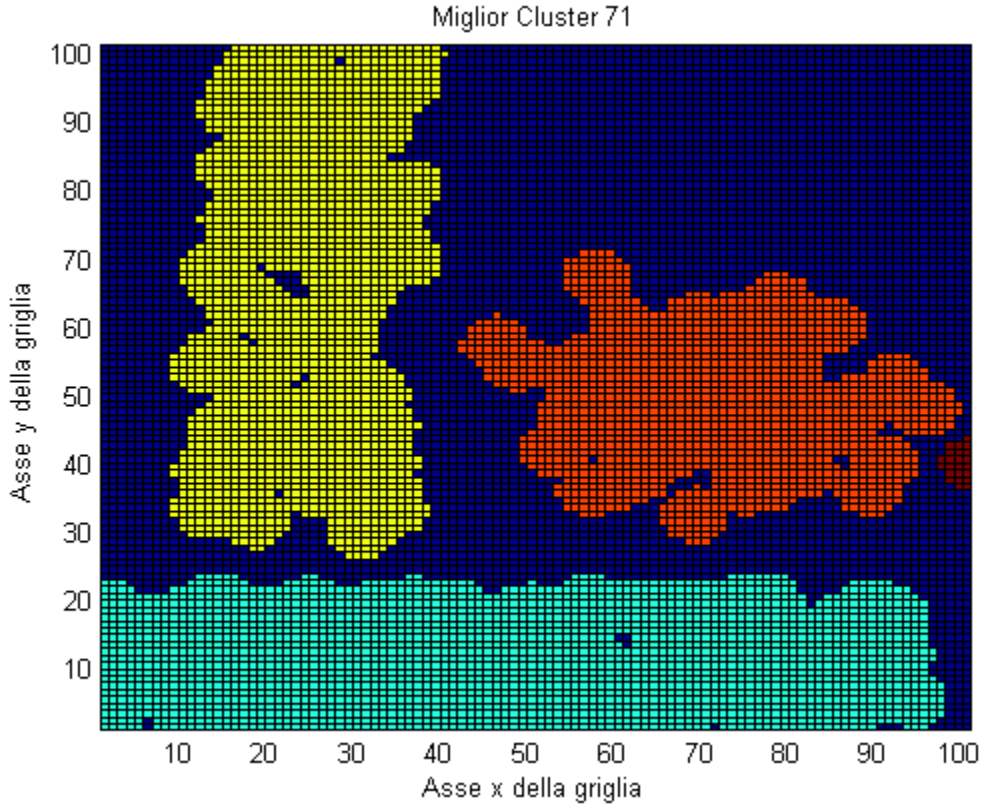
R15



SPIRAL


Miglior Cluster 72
Asse y della griglia
Asse x della griglia
10
20
30
40
50
60
70
80
90
100

LSUN

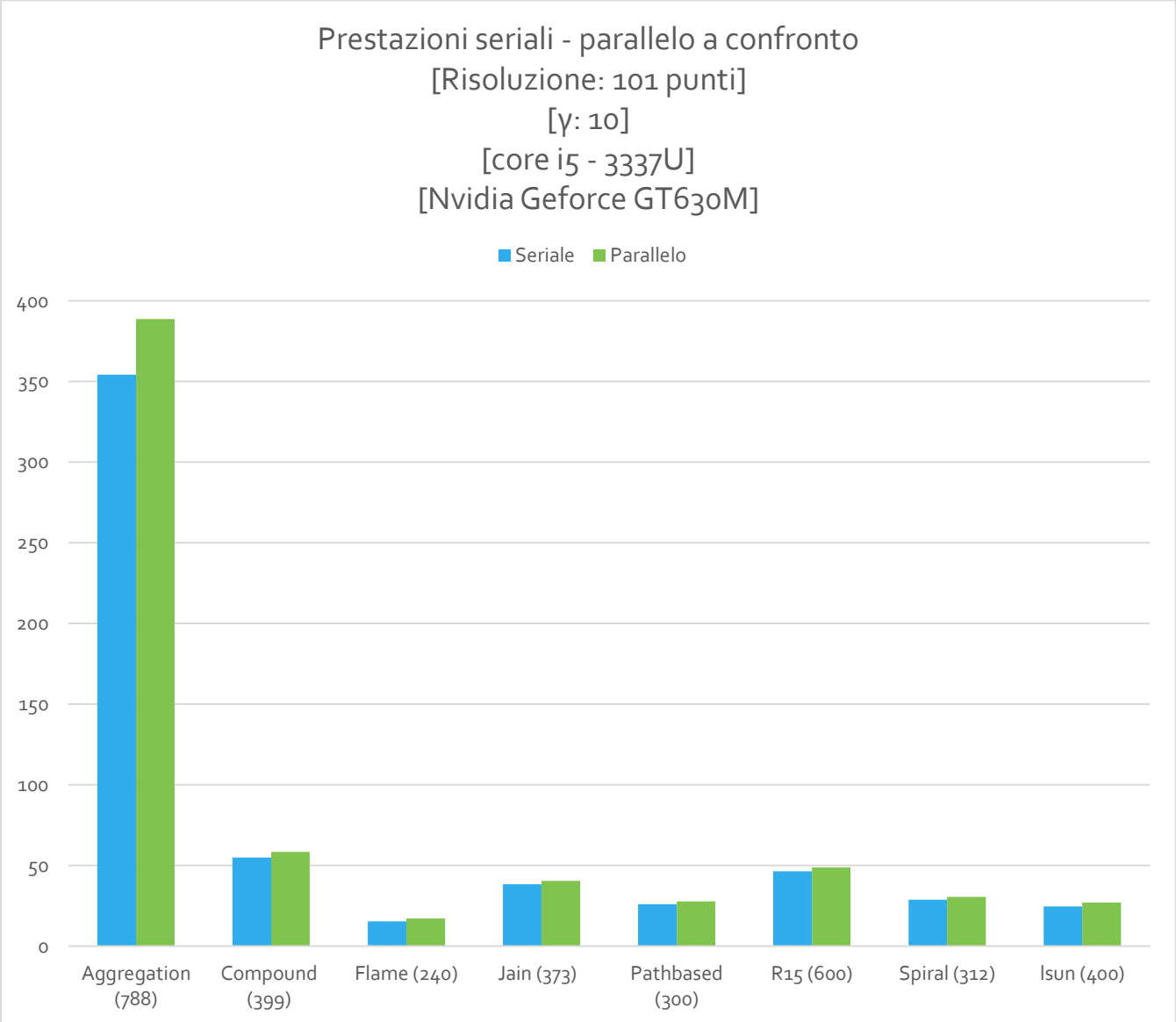

Miglior Cluster 71
Asse y della griglia
Asse x della griglia
10
20
30
40
50
60
70
80
90
100

Confronto tempi di esecuzione

Viene riportata una tabella che visualizza i differenti tempi di esecuzione dei due approcci, seriale e parallelo, su un sistema che monta i seguenti componenti:

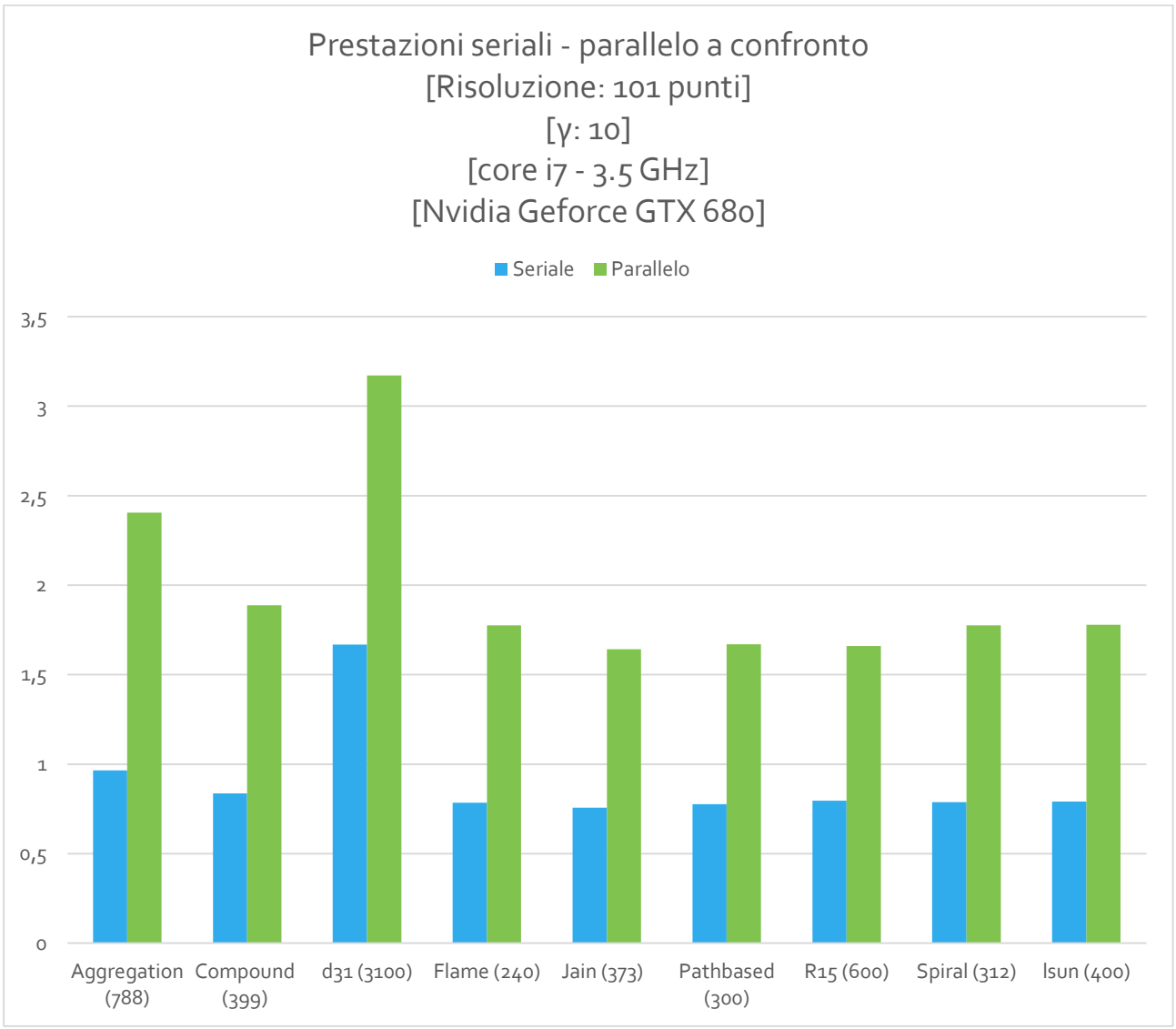
- Processore Intel Core i5-3337U @ 1.8 GHz – 2.5 GHz;
- 6 GB di Ram;
- Scheda video Nvidia Geforce GT630M
    - 96 cuda core;
    - 800 MHz clock grafico;
    - Cuda compute cabability 2.1;

I risultati sono stati i seguenti:



Vengono inoltre riportati i risultati di computazione *relativi al solo applicativo Visual Studio* (quindi manca la parte di elaborazione Matlab che si occupa della costruzione del dendrogramma); in questo caso i test

sono stati eseguiti con un sistema molto più performante che monta un processore core i7 con scheda video GTX 680 modificata con 12GB di memoria dedicata:



CONCLUSIONI

Ci sono due aspetti da evidenziare:

Il primo, è che da questo progetto non sono emersi chiari miglioramenti dall'utilizzo di un approccio parallelo rispetto ad uno seriale, e d'altronde questo aspetto era già stato messo in chiaro dal lavoro del collega G. Raimondi. Questo è dovuto al rallentamento introdotto dallo spostamento delle strutture dati tra memoria GPU e memoria di sistema.

Il secondo è che l'algortmo di clustering implementato, non risulta molto robusto in alcuni datasets, in cui si hanno agglomeramenti di cluster molto vicini. Questo perché l'algoritmo non utilizza un approccio con centroidi e distanze medie in cui si riesce a far fronte alla presenze di punti che sono una via di mezzo fra due cluster vicini, e quindi la definizione delle regioni di appartenenza dei cluster risultano delle volte imprecise.

Riferimenti

Connected-component labeling. (2014). Tratto da Wikipedia: http://en.wikipedia.org/wiki/Connected-component_labeling

Corp., N. (2014). *CUDA C PROGRAMMING GUIDE.* Tratto da Cuda toolkit documentation: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#abstract

Raimondi, G. (2014). *Parallelizzazione del calcolo della membership function di un dataset rispetto ad una griglia di punti N-dimensionale.* Roma - Latina.