David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Pràctica de Laboratori 1 - OpenMP

Anàlisi previ del codi:

Per a saber la part la qual hauríem de prioritzar a l'hora de realitzar millores al codi, vam seguir la llei d'Amdahl, i centrar-nos en la fracció de codi que més temps consumia en l'execució. Per a saber de quina es tractava, vam fer servir l'arxiu "job_gprof.sub", la qual executa i perfila el programa, i genera un informe ("informe_gprof.txt") indicant, per cada funció existent al codi, les vegades que es cridava i el temps total d'execució d'aquesta:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
74.10	4.72	**4.72**	**74649600**	0.00	**0.00**	**find_closest_centroid**
23.44	6.21	**1.49**	**1**	1.49	6.21	**kmeans**
2.05	6.34	**0.13**	**1**	0.13	0.13	**read_file**
0.63	6.38	**0.04**	**1**	0.04	0.04	**getChecksum**

Veiem en la taula generada que les funcions cridades un sol cop són les funcions de llegir l'arxiu, obtenir el checksum, i la funció principal de la llibreria (la qual crida a les anteriors citades).
La funció més cridada és "**find_closest_centroid**", que consumeix més d'un 74% de temps d'execució. Observem també que la funció, per cada crida, consumeix molt poc temps (4.72 segons/74649600 crides = 0.00000006 segons).
Determinem que el codi a canviar serà aquesta mateixa funció, o el bucle el qual la crida.
El codi de la funció **find_closest_centroid** és:

```
uint8_t find_closest_centroid(rgb* p, cluster* centroids, uint8_t num_clusters){
  uint32_t min = UINT32_MAX;
  uint32_t dis[num_clusters];
  uint8_t closest = 0, j;
  int16_t diffR, diffG, diffB;

  for(j = 0; j < num_clusters; j++)
  {
    diffR = centroids[j].r - p->r;
    diffG = centroids[j].g - p->g;
    diffB = centroids[j].b - p->b;
    // No sqrt required.
    dis[j] = diffR*diffR + diffG*diffG + diffB*diffB;

    if(dis[j] < min)
    {
      min = dis[j];
      closest = j;
    }
  }
  return closest;
```

Veiem que aquesta consta d'un bucle for per cada clúster, amb unes 3 restes, 3 multiplicacions, 2 sumes i un mètode simple per a determinar la distància mínima al clúster, per cada iteració. És important destacar que el número de clústers mai superarà els 64 en les nostres execucions. Tenint en compte això, distribuir i paral·lelitzar el treball del bucle entre fils per a cada crida (74649600 cops), no resultaria la millor opció per optimitzar la funció. Atès que la funció és relativament simple i amb poc marge de millora, decidim ajustar el codi (bucle) que la crida, i procedir a paral·lelitzar en la mesura del possible aquestes crides.

Anàlisi del codi a millorar:

El bucle en qüestió que hem decidit paral·lelitzar donades les característiques de l'execució i del codi és el següent (a part de la declaració i inicialització de cada variable):

```
// Find closest cluster for each pixel
for(j = 0; j < num_pixels; j++)
{
   closest = find_closest_centroid(&pixels[j], centroides, k);
   centroides[closest].media_r += pixels[j].r;
   centroides[closest].media_g += pixels[j].g;
   centroides[closest].media_b += pixels[j].b;
   centroides[closest].num_puntos++;
}
```

Comprovem que la raó de tal número de crides és que aquest bucle executa la funció "find_closest_centroid" tantes vegades com píxels en la imatge. Això ho fa per a cada iteració de la funció kmeans.
La primera instrucció dins el bucle és la obtenció d'una variable "closest", que indica el clúster més proper al píxel d'aquella iteració. A partir de la variable "closest", obtenim el número de clúster que es fa servir en les següents 4 instruccions, que són un sumatori per cada color i nombre de pixels en el centroide corresponent. Aquestes obtenen el valor de cada color del píxel de la iteració actual, i li sumen a la mitjana del clúster al qual pertanyen. És a dir, que les quatre últimes instruccions depenen de la primera (per a assignar el clúster a aplicar el sumatori).
És important destacar que, per a cada iteració, la variable "closest" no serà (necessàriament) la mateixa.

Estratègia de millora:

La nostra estratègia és la de crear una secció paralela de tipus for, amb les següents restriccions:

- Privatitzar la variable "closest". A l'anàlisi del codi s'esmenta que aquesta variable depèn de cada píxel donada la funció "find_closest_centroid". Donada aquesta dependència, hem de dedicar una variable "closest" a cada fil per tal d'evitar conflictes en l'accés de posicions de la variable "centroides[]".
- Aplicar una reducció per les variables dins de cada element centroides[closest] com poden ser "centroides[closest].media_r" o "centroides[closest].num_puntos". Per

cada fil, s'acumula un valor en aquestes variables, i per a una correcta execució, hem de "sumar els sumatoris" per cada fil.

A part de fer aquest canvi, podríem també paral·lelitzar bucles simples que iteren per cada clúster (generalment poques iteracions), però donat el nombre tan reduït d'iteracions, no ho veiem necessari.

Aplicació de la paral·lelització:

Clàusules d'OpenMP:

- La clàusula del bucle és: **#pragma omp parallel for.** Repartim equitativament el nombre d'iteracions per cada fil (no indiquem l'schedule, per tant, s'aplica el static per defecte)
- En el nostre cas, no indiquem el nombre de fils (**num_threads**) com a clàusula explícita, ja que farem servir el codi "export OMP_NUM_THREADS=8" dins l'arxiu job.sub per a canviar el nombre de threads més fàcilment sense accedir al codi.
- Per a privatitzar la variable "closest" fem servir "**private(closest)**". No ens interessa fer servir firstprivate ni lastprivate ja que aquesta variable no té un valor definit abans d'entrar al bucle ni volem el valor final, ja que només la fem servir per a accedir a una posició de "centroides[]"
- Reduction: per tal de combinar els resultats de cada fil i sumar-los al final, apliquem una reducció. Però amb el mètode per defecte d'OpenMP, no accepta la sintaxis com "reduction(+:centroides[closest].media_r), ja que accedim a un atribut d'un element d'un array, on es guarda la suma resultant.
  Per no haver de programar un mètode de reducció a mida pel bucle, decidim simplificar aquesta variable, creant-ne quatre: Red[i], green[i], blue[i] i points[i], que equivaldrien a les variables centroides[i].media_r, centroides[i].media_g, centroides[i].media_b, centroides[i].num_puntos, respectivament (s'han de declarar i inicialitzar a 0),
  Amb aquest canvi fet, la clàusula resultant és "**reduction(+:red[:k], green[:k], blue[:k], points[:k])**".
  Després de l'execució del bucle, caldrà fer-ne un altre per a assignar, a cada centoide, els valors del sumatori per a continuar l'execució.

Amb aquestes clàusules i canvis, el codi resultant és el següent:

```
// Find closest cluster for each pixel
//Inicialitzem les variables auxiliars que farem servir amb calloc()
//Tots els valors per cada cluster a 0 (R, G, B i points)
uint32_t* red = calloc(k, sizeof(uint32_t));
uint32_t* green = calloc(k, sizeof(uint32_t));
uint32_t* blue = calloc(k, sizeof(uint32_t));
uint32_t* points = calloc(k, sizeof(uint32_t));

// Sense clàusula num_threads(X) ja que la canviem a l'arxiu job.sub amb: export
OMP_NUM_THREADS=2
//Calculem els valors de les variables auxiliars donats els clústers més propers per cada píxel
#pragma omp parallel for private(closest) reduction(+:red[:k], green[:k], blue[:k], points[:k])
for(j = 0; j < num_pixels; j++)
{
   closest = find_closest_centroid(&pixels[j], centroides, k);
   red[closest] += pixels[j].r;
   green[closest] += pixels[j].g;
   blue[closest] += pixels[j].b;
   points[closest]++;
}

// Actualitzem els valors dels centroides (k iteracions, no cal paral·lelitzar)
for(i = 0; i < k; i++)
{
   centroides[i].media_r = red[i];
   centroides[i].media_g = green[i];
   centroides[i].media_b = blue[i];
   centroides[i].num_puntos = points[i];
}
```

Mètriques obtingudes:

Fent servir la comanda “perf stat ./kmeans_OpenMP.exe test k imagen.bmp” on k són els centroides, i prèviament seleccionant el nombre de fils amb “export OMP_NUM_THREADS=8”, hem elaborat unes taules per a visualitzar els valors de temps, speedup, i eficiència segons valors de k d’entre 2 i 16 i valors de threads d’entre 1 i 8. Hem utilitzat el codi original per a medir el temps en el cas de les files amb un sol fil. Per una millor comparació visual, hem canviat el color de les cel·les segons el valor d’aquestes: Verd → Millor valor, Vermell → Pitjor valor:

David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Time elapsed (s)

Threads \ Valor de K (clústers)	2	4	8	16
1	1.30	4.54	19.16	34.75
2	0.81	2.44	9.46	16.34
4	0.49	1.31	4.93	8.43
8	0.31	0.73	2.58	4.39

Observem, com era d'esperar, que com a més clústers i menys fils, major és el temps d'execució. El valor mínim (0.31 segons) l'hem obtingut amb 8 fils i k=2, mentre que el màxim ha sigut amb el codi original (1 fil) i k=16.

Speedup (respecte original)

Threads \ Valor de K (clústers)	2	4	8	16
1	1.00	1.00	1.00	1.00
2	1.60	1.86	2.03	2.13
4	2.65	3.47	3.89	4.12
8	4.19	6.22	7.43	7.92

En aquest gràfic podem observar realment l'acceleració que obtenim depenent del nombre de fils. Veiem que, com a més clústers, més d'apropa al valor de l'speedup al nombre de fils. En alguns casos veiem que l'speedup supera el nombre de fils, en principi degut a variacions en l'execució i en l'estat del servidor.

Eficiència

Threads \ Valor de K (clústers)	2	4	8	16
1	1.00	1.00	1.00	1.00
2	0.80	0.93	1.01	1.06
4	0.66	0.87	0.97	1.03
8	0.52	0.78	0.93	0.99

En aquest gràfic podem visualitzar l'eficiència de les execucions. Observem com, a mida que augmenta el nombre de fils, l'eficiència és més propera a 1 (valor teòric òptim). Això és degut a que, dediquem més còmput a les funcions com la que hem millorat (part paral·lelitzable), en relació a les parts que s'han d'executar en seqüencial, com la lectura de l'imatge o el càlcul del checksum. Això és un molt bon indicador de l'escalabilitat de la funció: si seguim augmentant el nombre de nuclis, el valor de l'speedup s'aproparà cada cop més al nombre de nuclis (és a dir, l'eficiència serà més pròxima a 1). Això també ens indica que hem millorat la part correcta del codi, la que consumia més temps, i que l'anàlisi del codi ha estat el correcte.