

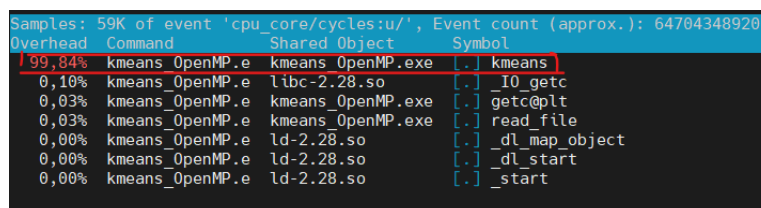
# Presentación Laboratorio

## Paralelización programa

### Análisis código secuencial

El objetivo de este laboratorio es paralelizar el código del programa `kmeans`. Para ello, hemos hecho un análisis general del rendimiento.

Con la consulta `perf record ./kmeans_OpenMP.exe test 32 imagen.bmp` hemos obtenido una visión detallada del consumo de tiempo de cada función del programa `kmeans`. El resultado obtenido es la siguiente tabla:



Overhead	Command	Shared Object	Symbol
99.84%	kmeans_OpenMP.e	kmeans_OpenMP.exe	[.] kmeans
0.10%	kmeans_OpenMP.e	libc-2.28.so	[.] _IO_getc
0.03%	kmeans_OpenMP.e	kmeans_OpenMP.exe	[.] getc@plt
0.03%	kmeans_OpenMP.e	kmeans_OpenMP.exe	[.] read_file
0.00%	kmeans_OpenMP.e	ld-2.28.so	[.] _dl_map_object
0.00%	kmeans_OpenMP.e	ld-2.28.so	[.] _dl_start
0.00%	kmeans_OpenMP.e	ld-2.28.so	[.] _start

Determinamos que el bottleneck del programa es la función `kmeans()`, que se encuentra en el archivo `kmeanslib.c`, con un 99.84% del consumo del rendimiento. En la función, hay 4 bucles for. El primero se utiliza para inicializar los centroides. El segundo para reiniciarlos. El tercer bucle sirve para encontrar el clúster más cercano para cada píxel y el último para actualizar los centroides de cada clúster.

Todos los bucles, excepto el tercero, dependen del parámetro `k`. A diferencia del resto, depende del número de píxeles de la imagen. Dado que `k` es significativamente menor que el número de píxeles, paralelizaremos únicamente el tercer bucle for, que encuentra el clúster más cercano para cada píxel. Aunque podríamos paralelizar los otros tres bucles, estos no tendrán tanto impacto en la mejora del rendimiento ya que, a comparación con el tercer bucle, implican un mucho menor número de iteraciones y operaciones.

Si nos centramos en el bucle mencionado (find closest cluster), en cada iteración, llama a la función `find_closest_centroid()`. Esta contiene un bucle que se realiza `k` veces. Por lo tanto, se ejecutan `k * píxeles` iteraciones.

### Paralelización OpenMP

Para poder mejorar el rendimiento del programa, hemos utilizado la librería OpenMP. Tal y como hemos dicho, hemos paralelizado el bucle de la función `kmeans()`.

Lo primero que hemos hecho ha sido crear una región paralela con `#pragma omp parallel for`. Debido a que no hay ninguna dependencia de datos, podemos paralelizar completamente el bucle. Como nuestro objetivo es hacer la reducción de: `centroides[closest].media_r += pixels[j].r`, necesitamos crear unos vectores auxiliares. Esto debe de ser así ya que la cláusula `reduction` no puede acceder a la estructura interna de un struct para realizar este tipo de operaciones. Hemos elegido esta cláusula porque realiza las sumas de forma eficiente:  $O(\log 2)$ .

Por lo tanto, hemos creado los vectores auxiliares: `media_r`, `media_g`, `media_b` y `num_puntos`. Para ello, primero hemos reservado memoria utilizando la función `malloc()` e inicializado en el bucle que inicializa los centroides. Cabe destacar que estos vectores tienen un tamaño `k`, es decir, tantas posiciones como número de centroides. Por lo que cada posición corresponde a la media de valores de un clúster.

Con estas modificaciones, nos aseguramos de que, por cada iteración del bucle paralelizado, accede únicamente a su propio píxel y centroide. Por lo tanto, no se aprecia ninguna dependencia de datos entre las iteraciones del bucle paralelizado.

Posteriormente hemos paralelizado el bucle donde se llama la función `find_closest_centroid()`. A diferencia del bucle de la función, que calcula el centroide más cercano por un solo píxel, se calcula por todos los píxeles añadiendo un bucle `for` externo que itera por cada píxel. Los resultados se guardan en un vector auxiliar. Este bucle se paraleliza mediante el uso de `#pragma omp parallel for`. De esta manera cada thread realiza esta tarea de manera independiente, lo que permite que múltiples threads trabajen simultáneamente en diferentes píxeles de la imagen. Es decir, se calcularán de forma simultánea los centroides más cercanos de diferentes píxeles.

Al modificar tanto la estructura de la función `find_closest_centroid()`, se ha decidido no llamar a esta, y escribir el código que realiza este cálculo de forma paralela dentro de la función `kmeans()`. De esta manera, se ha conseguido que el código utilice más recursos de la CPU mediante la ejecución simultánea en diferentes núcleos para los diferentes threads, resultando en un código más eficiente.

## Código paralelizado OpenMP

Finalmente, el código paralelizado es el siguiente:

```
/*
 * Main function k-means
 * input @param: K --> number of clusters
 * input @param: centroides --> the centroids
 * input @param: num_pixels --> number of total pixels
 * input @param: pixels --> pointer to array of rgb (pixels)
 */
void kmeans(uint8_t k, cluster* centroides, uint32_t num_pixels, rgb* pixels){
    uint8_t condition, changed, closest;
    uint32_t i, j, random_num;

    //Reservem memòria
    uint32_t* media_r = malloc(k * sizeof(uint32_t));
    uint32_t* media_g = malloc(k * sizeof(uint32_t));
    uint32_t* media_b = malloc(k * sizeof(uint32_t));
    uint32_t* num_puntos = malloc(k * sizeof(uint32_t));

    uint32_t* closest_centroide = malloc(num_pixels*sizeof(uint32_t));

    //Inicialitzem vectors i variables auxiliars
    uint32_t sum_media_r = 0, sum_media_g = 0, sum_media_b = 0, sum_num_puntos = 0;
```

```

printf("STEP 1: K = %d\n", k);
k = MIN(k, num_pixels);

// Init centroids
printf("STEP 2: Init centroids\n");
for(i = 0; i < k; i++)
{
    random_num = rand() % num_pixels;
    centroides[i].r = pixels[random_num].r;
    centroides[i].g = pixels[random_num].g;
    centroides[i].b = pixels[random_num].b;
}

// K-means iterative procedures start
printf("STEP 3: Updating centroids\n\n");
i = 0;
do
{
    // Reset centroids i vectores auxiliares
    for(j = 0; j < k; j++)
    {
        centroides[j].media_r = 0;
        centroides[j].media_g = 0;
        centroides[j].media_b = 0;
        centroides[j].num_puntos = 0;
        media_r[j] = 0;
        media_g[j] = 0;
        media_b[j] = 0;
        num_puntos[j] = 0;
    }

    // Find closest cluster for each pixel
    #pragma omp parallel for
    for(j=0; j<num_pixels; j++)
    { //Per cada pixel,
        int16_t diffR, diffG, diffB;
        rgb* p=&pixels[j];
        uint32_t min = UINT32_MAX, distancia, centroide_actual;

        for (centroide_actual=0; centroide_actual<k; centroide_actual++){
            diffR = centroides[centroide_actual].r - p->r;
            diffG = centroides[centroide_actual].g - p->g;
            diffB = centroides[centroide_actual].b - p->b;
            distancia = diffR*diffR + diffG*diffG + diffB*diffB;

            if (distancia < min){
                min = distancia;
                closest_centroide[j] = centroide_actual;
            }
        }
    }
}

```

```
//printf("Centroides encontrados");
// Assign colors to each cluster
#pragma omp parallel for
reduction(+:media_r[:k],media_g[:k],media_b[:k],num_puntos[:k])
for(j = 0; j < num_pixels; j++)
{
    uint32_t closest_cluster = closest_centroide[j];
    media_r[closest_cluster] += pixels[j].r;
    media_g[closest_cluster] += pixels[j].g;
    media_b[closest_cluster] += pixels[j].b;
    num_puntos[closest_cluster] += 1;
}

// Update structs
for (j=0; j<k; j++){
    centroides[j].media_r = media_r[j];
    centroides[j].media_g = media_g[j];
    centroides[j].media_b = media_b[j];
    centroides[j].num_puntos = num_puntos[j];
}

// Update centroids & check stop condition
condition = 0;
for(j = 0; j < k; j++)
{
    if(centroides[j].num_puntos == 0)
    {
        continue;
    }

    centroides[j].media_r =
centroides[j].media_r/centroides[j].num_puntos;
    centroides[j].media_g =
centroides[j].media_g/centroides[j].num_puntos;
    centroides[j].media_b =
centroides[j].media_b/centroides[j].num_puntos;
    changed = centroides[j].media_r != centroides[j].r ||
centroides[j].media_g != centroides[j].g || centroides[j].media_b != centroides[j].b;
    condition = condition || changed;
    centroides[j].r = centroides[j].media_r;
    centroides[j].g = centroides[j].media_g;
    centroides[j].b = centroides[j].media_b;
}

i++;
} while(condition);
printf("Number of K-Means iterations: %d\n\n", i);
}
```

## Análisis ejecución

Una vez paralelizado el código de kmeanslib.c, hemos estudiado el rendimiento de cada programa en las diferentes máquinas: Aolin y Wilma.

Los tiempos de ejecución obtenidos en Aolin son los siguientes:

AOLIN						
Tiempos de referencia - Versión Secuencial						
Versión: Seq -Ofast	K=2	K=4	K=8	K=16	K=32	K=64
Elapsed time	0,27	1,06	4,26	7,32	15,32	121,85
Iteraciones	9	22	51	47	49	196
Checksum	6405336	12689895	22329779	42559141	73490886	123269810

Tiempos de referencia - Versión Paralela							
Nº threads	Versión: OpenMP -Ofast	K=2	K=4	K=8	K=16	K=32	K=64
2	Elapsed time	0,37	0,63	3,32	3,79	8,48	64
4		0,12	0,37	1,34	2,13	4,63	34,72
6		0,11	0,27	0,97	1,5	3,14	23,24
8		0,1	0,29	1,06	1,86	3,9	29,04
10		0,09	0,25	0,92	1,48	3,25	24,18
12		0,08	0,22	0,78	1,28	2,71	20,64
Iteraciones		9	22	51	47	49	196
Checksum		6405336	12689895	22329779	42559141	73490886	124269810

Speedup						
Nº threads	K=2	K=4	K=8	K=16	K=32	K=64
2	0,73	1,68	1,28	1,93	1,81	1,90
4	2,25	2,86	3,18	3,44	3,31	3,51
6	2,45	3,93	4,39	4,88	4,88	5,24
8	2,70	3,66	4,02	3,94	3,93	4,20
10	3,00	4,24	4,63	4,95	4,71	5,04
12	3,38	4,82	5,46	5,72	5,65	5,90

Eficiencia						
Nº threads	K=2	K=4	K=8	K=16	K=32	K=64
2	0,36	0,84	0,64	0,97	0,90	0,95
4	0,56	0,72	0,79	0,86	0,83	0,88
6	0,41	0,65	0,73	0,81	0,81	0,87
8	0,34	0,46	0,50	0,49	0,49	0,52
10	0,30	0,42	0,46	0,49	0,47	0,50
12	0,28	0,40	0,46	0,48	0,47	0,49

Como podemos observar, para todas las ejecuciones se ha obtenido el mismo número de iteraciones y checksums. Por lo tanto, el programa de la versión de OpenMP -Ofast se ha paralelizado de forma correcta. Además, verificamos que los resultados son correctos y podemos analizarlos exhaustivamente.

La aceleración del programa aumenta conforme incrementamos el número de recursos. Al agregar threads y mantener el número de clústers (k), se intensifica el speedup. Por ejemplo, observemos k=64. Cuando hay 2 threads, la aceleración es de 1,90 respecto el programa secuencial. En cambio, cuando hay 12 threads, el speedup es de 5,90. Es decir, la aceleración se triplica:  $\frac{5,90}{1,90} = 3,11x$ .

Además, la aceleración del programa también aumenta conforme incrementamos el número de clústers (k). Si mantenemos los threads, por ejemplo 10, vemos que cuando k=2 el programa paralelo se ejecuta tres veces más rápido que el secuencial, ya que el speedup es de 3x. Si k=64, se ejecuta casi seis veces más rápido. Por lo tanto, el paralelismo es mayor conforme k aumenta.

Si analizamos la tabla de eficiencia, observamos que esta mejora al augmenar k. Cuando threads=6 i k=2, hay una eficiencia de 0,41. Al establecer k=64, esta es de 0,87. Por lo tanto, mejora un  $\frac{0,87}{0,41} = 2,12x$  (el doble).

La eficiencia mide qué tan bien se está utilizando cada recurso adicional. Si la eficiencia es próxima a 1, significa que cada recurso está siendo utilizado de forma efectiva y contribuye significativamente al rendimiento total. Si es mejor que 1, esto no es así y, por lo tanto, los recursos se utilizan de manera ineficiente. Si observamos la tabla, a partir de los 6 threads la eficiencia cae en picado, siendo esta como máximo un 0,52. Observamos que cuando k=64 la eficiencia mayor es al utilizar dos threads con un 0,95 y la menor al utilizar 12 threads con una eficiencia de 0,49. Vemos que el esta decrece un  $\frac{0,49}{0,95} = 0,5x$ , es decir, el doble de ineficiente.

Los tiempos de ejecución obtenidos con Wilma son los siguientes:

WILMA						
Tiempos de referencia - Versión Secuencial						
Versión: Seq -Ofast	K=2	K=4	K=8	K=16	K=32	K=64
Elapsed time	1,43	4,56	19,16	34,77	69,87	530,09
Iteraciones	9	22	51	47	49	196
Checksum	6405336	12689895	22329779	42559141	73490886	124269810

Tiempos de referencia - Versión Paralela							
Nº threads	Versión: OpenMP -Ofast	K=2	K=4	K=8	K=16	K=32	K=64
2	Elapsed time	0,88	2,52	10,17	17,78	35,87	267,13
4		0,53	1,39	5,28	9,15	17,93	132,9
6		0,41	0,98	3,62	6,25	12,15	89,56
8		0,34	0,77	2,76	4,77	9,24	67,94
10		0,42	0,69	2,27	3,86	7,44	54,58
12		0,3	0,59	1,96	3,32	6,47	45,96
Iteraciones		9	22	51	47	49	196
Checksum		6405336	12689895	22329779	42559141	73490886	124269810

Speedup						
Nº threads	K=2	K=4	K=8	K=16	K=32	K=64
2	1,63	1,81	1,88	1,96	1,95	1,98
4	2,70	3,28	3,63	3,80	3,90	3,99
6	3,49	4,65	5,29	5,56	5,75	5,92
8	4,21	5,92	6,94	7,29	7,56	7,80
10	3,40	6,61	8,44	9,01	9,39	9,71
12	4,77	7,73	9,78	10,47	10,80	11,53

Eficiencia						
Nº threads	K=2	K=4	K=8	K=16	K=32	K=64
2	0,81	0,90	0,94	0,98	0,97	0,99
4	0,67	0,82	0,91	0,95	0,97	1,00
6	0,58	0,78	0,88	0,93	0,96	0,99
8	0,53	0,74	0,87	0,91	0,95	0,98
10	0,34	0,66	0,84	0,90	0,94	0,97
12	0,40	0,64	0,81	0,87	0,90	0,96

Gracias al análisis anterior, hemos confirmado que la paralelización es correcta. Por lo tanto, los resultados son correctos y podemos analizarlos exhaustivamente.

De igual forma que en Aolin, en Wilma la aceleración del programa aumenta conforme incrementamos el número de recursos. Al agregar threads y mantener el número de clústers (k), se intensifica el speedup. Por ejemplo, observemos k=64. Cuando hay 2 threads, la aceleración es de 1,98 respecto el programa secuencial. En canvio, cuando hay 12 threads, el speedup es de

11,53. Es decir, la aceleración se multiplica por seis:  $\frac{11,53}{1,98} = 5,82x$ . Si comparamos el rendimiento respecto máquinas, Wilma duplica la aceleración del programa respecto a Aolin ya que la diferencia de speedups es:  $\frac{5,82}{3,11} = 1,87x$ .

Similar al Aolin, en Wilma la aceleración del programa también aumenta conforme incrementamos el número de clústers (k). Si mantenemos los threads, por ejemplo 10, vemos que cuando k=2 el programa paralelo se ejecuta tres veces más rápido que el secuencial, ya que el speedup es de 3.4x. Si k=64, se ejecuta casi diez veces más rápido. Por lo tanto, el paralelismo es mayor conforme k aumenta:  $\frac{9,71}{3,40} = 2,85x$ . De nuevo, Wilma genera una mayor aceleración que Aolin, concretamente un  $\frac{2,85}{2,12} = 1,34x$ .

Si analizamos la tabla de eficiencia, observamos que esta mejora al augmenar k. Cuando threads=6 i k=2, hay una eficiencia de 0,58. Al establecer k=64, esta es de 0,99. Por lo tanto, mejora un  $\frac{0,99}{0,58} = 1,71x$ . Si comparamos las eficiencias de las dos máquinas, al tener un gran número de clusters, como k=64, la eficiencia es aproximadamente 1. No solo eso, sino que al aumentar el número de threads no decrece, se mantiene estable y próxima al máximo. Recordemos que esto con Aolin no pasaba y que al llegar a threads=6, la eficiencia disminuía.

## Conclusiones

Al paralelizar el código kmeanslib.c hemos aprendido cómo usar correctamente las cláusulas de la librería OpenMP. No solo eso, hemos sido conscientes de que a veces no basta con paralelizar los bucles añadiendo pragmas y cláusulas. Muchas veces es necesario reescribir el código, como hemos hecho con la función *find\_closest\_centroid()*.

Además, hemos ejecutado el código paralelo modificando los recursos disponibles y el tamaño del problema. Al analizar los resultados, hemos verificado que la paralelización es correcta y hemos determinado que el mismo programa no es escalable en las dos máquinas disponibles: Aolin y Wilma.

En Aolin, kmeanslib.c paralelizado con OpenMP no es escalable ya que cuando los threads disponibles son mayores a 6, la eficiencia decrece significativamente. Esto sucede porque Aolin tiene 6 cores, y 2 threads por core. Por lo tanto, no puede aprovechar el multithread. Esto no sucede en Wilma, la cual mantiene una eficiencia próxima a 1. A diferencia de Aolin, tiene 12 cores, por lo que sí puede aprovechar el multithread. Es por eso por lo que sus mejoras en la aceleración son mayores, porque los recursos están mejor aprovechados y favorecen notablemente al rendimiento total. Por lo tanto, el programa sí es escalable en Wilma, porque no decrece la eficiencia al aumentar el tamaño del problema y los cores utilizados.