Computación de altas prestaciones

Implementación de kmeans:

Serie -> MPI -> OpenMP

Francisco Jesús Díaz Pellejero

Javier Villar Asensio

Serie 2
Pasos del algoritmo 2
Medidas de tiempo 3
MPI 7
Introducción 7
Implementación 7
Repartir la imagen 8
Repartir las etiquetas 8
Compartir los centroides 9
Ejecución serial de todos los procesos 9
Creación de nuevos centroides 10
Recoger resultados y ver convergencia 10
Resultado de la ejecución 12
OpenMP 12
Introducción 12
Implementación 13
Inicio 13
Bucle principal 13
Asignar puntos a los centroides 13
Actualizar los centroides 15
Comprobar la convergencia 15

Serie

Pasos del algoritmo

1. Inicialización de centros: Para inicializar los centroides hemos utilizado el método aleatorio, en este, escogemos puntos dentro del dataset de forma aleatoria y asignamos un centroide en los puntos. El número de centroides viene dado por el parámetro K de Kmeans y la localización de cada uno se calcula mediante la función *uniform* de la clase RNG (random number generator) de opencv.

Con los centroides inicializados podemos empezar las iteraciones del algoritmo para realizar la agrupación. Los criterios que hemos utilizado para dar como completada la ejecución son la cantidad de iteraciones y una épsilon. El número máximo de iteraciones marcará la cantidad de iteraciones que va a realizar antes de que se fuerce el fin de la ejecución, de este modo se puede parar cuando esté atrapado en bucles en los que la posición de los centroides no converja. La épsilon será el

indicador con el que sabremos que el cambio de localización entre iteraciones de los centroides es tan pequeño que está convergiendo a un valor, es decir, que si la distancia más alta de las de los centroides entre dos iteraciones es menor a la épsilon asumimos que sus localizaciones convergen y puede finalizar la ejecución. Tanto la épsilon como la cantidad de iteraciones se han englobado en un tipo *TermCriteria* de opencv.

2. Asignamos los puntos a los clusters que les correspondan: Calculamos la distancia euclídea de cada punto a todos los clusters y asignamos en la matriz de etiquetas el cluster más cercano al punto. La distancia euclídea se ha calculado mediante el método *norm* de opencv y viene determinada por la siguiente formula:

   sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y))

   Esta fórmula equivale a:

   sqrt((a - b).x * (a - b).x + (a - b).y * (a - b).y)

   Por lo tanto, podemos llamar a la función de la siguiente manera para así calcular la distancia euclídea:

   cv::norm(a - b)

3. Ya asignados todos los puntos a un cluster actualizamos las posiciones de los centroides según los nuevos datos.
4. Calculamos la distancia de los nuevos centroides con los anteriores hasta encontrar una convergencia. En caso de no alcanzar el valor de épsilon y no alcancemos el número de intentos, se repite la iteración del algoritmo.

Medidas de tiempo

Para la toma de tiempo de cómputo de nuestro algoritmo se ha utilizado la función "std::chrono::high_resolution_clock" de la librería <chrono>. Esta se basa en el reloj de alta resolución del sistema por lo que se tiene una precisión mucho mayor que la función "clock()" estándar, que generalmente se resuelve en milisegundos.

Se medirán los tiempos del algoritmo en su conjunto retocando hiperparámetros para después medir cuánto tarda cada parte de este.

Para todas las mediciones se tomará el fichero pavia.txt con 1096x715 pixeles con 102 componentes por punto.

Number of Clusters	Attempts	Epsilon	Time Taken
3	10	0.2	10 sec 277 ms
3	10	0.5	9 sec 424 ms

3	10	1	9 sec 995 ms
3	50	0.2	44 sec 675 ms
3	50	0.5	56 sec 904 ms
3	50	1	39 sec 138 ms
3	100	0.2	44 sec 569 ms
3	100	0.5	56 sec 503 ms
3	100	1	39 sec 246 ms
5	10	0.2	15 sec 203 ms
5	10	0.5	14 sec 724 ms
5	10	1	14 sec 774 ms
5	50	0.2	93 sec 473 ms
5	50	0.5	76 sec 521 ms
5	50	1	94 sec 247 ms
5	100	0.2	137 sec 10 ms
5	100	0.5	126 sec 533 ms

5	100	1	119 sec 623 ms
7	10	0.2	31 sec 108 ms
7	10	0.5	20 sec 129 ms
7	10	1	20 sec 282 ms
7	50	0.2	133 sec 265 ms
7	50	0.5	104 sec 166 ms
7	50	1	120 sec 518 ms
7	100	0.2	164 sec 596 ms
7	100	0.5	146 sec 704 ms
7	100	1	133 sec 705 ms
10	10	0.2	27 sec 417 ms
10	10	0.5	27 sec 179 ms
10	10	1	27 sec 171 ms
10	50	0.2	140 sec 80 ms
10	50	0.5	140 sec 606 ms

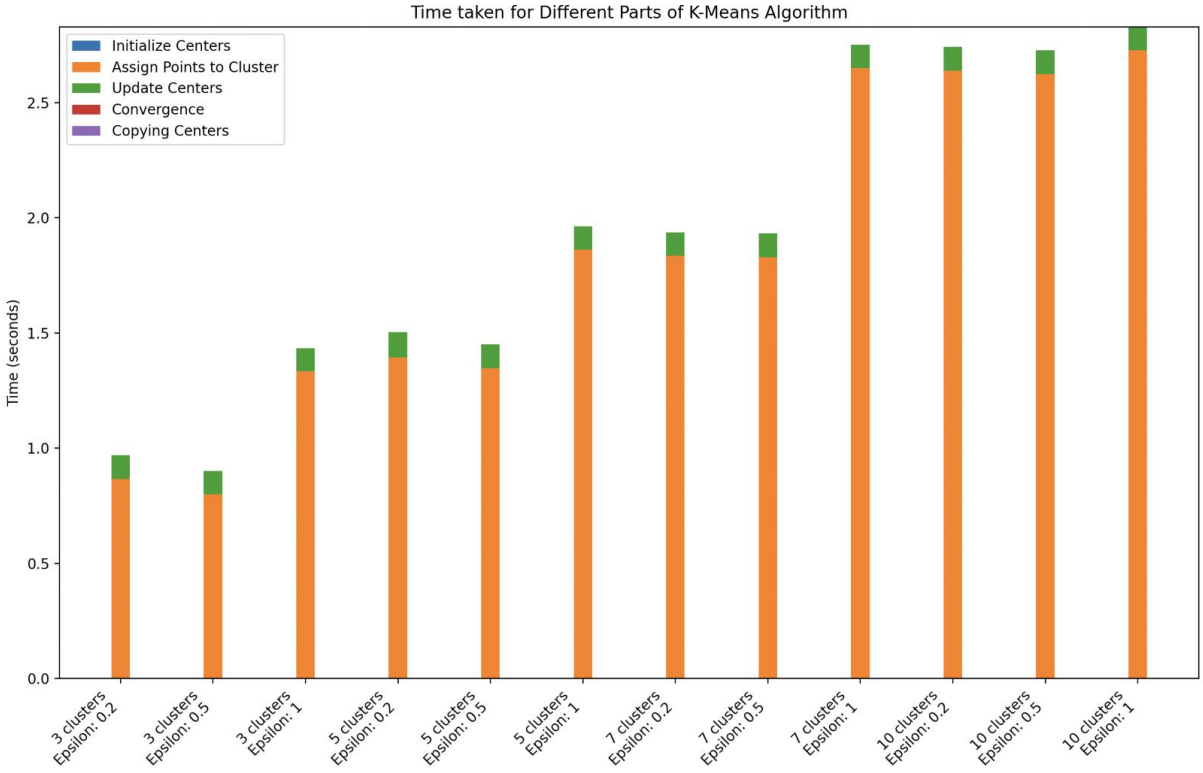
10	50	1	139 sec 585 ms
10	100	0.2	283 sec 553 ms
10	100	0.5	283 sec 300 ms
10	100	1	289 sec 435 ms

El valor de épsilon no influye de manera significativa en el tiempo de ejecución del algoritmo. Con 0.2 se incrementa el tiempo de cómputo al necesitar unos centroides más cercanos unos de otros (a excepción de un par de casos) pero los factores más determinantes son el número de clusters y los intentos a ejecutar. En el gráfico de arriba se muestra un gráfico para una épsilon con valor 0.2.

Sin embargo, estos tiempos de ejecución pueden variar dependiendo del equipo en que se ejecuten y otros aspectos que dependan del momento en que se ejecutan (nivel de batería, cantidad de procesos activos, ...). En otro equipo, en concreto un Intel(R) Core(TM) i5-6200U con 4 núcleos y 4

GB de memoria RAM, estas mismas ejecuciones tardan entre un 2.5 a 3 veces más que los tiempos mostrados anteriormente (Apple M2 con 8 núcleos y 16GB de RAM).

A continuación, se muestra el tiempo de cada una de las partes del algoritmo para un solo intento:



Como se puede apreciar en el gráfico. La inicialización de los centros y la copia del array de centros antiguos al array de centros nuevos supone un tiempo despreciable. La mayoría del tiempo del algoritmo está dedicado a la asignación de puntos a los distintos clusters, incluyendo así el cálculo de la distancia entre los distintos puntos.

MPI

Introducción

Tras haber implementado k-means con un algoritmo en serie, queremos estudiar la mejora en velocidad que supondría paralelizarla, primero con un paradigma de paso de mensajes. Se utiliza para el paso de mensajes la librería para C/C++ que se convirtió en un estándar de facto.

Aunque repartimos la carga computacional entre varios procesadores, en el intercambio de mensajes añade una sobrecarga, así que se pretende ver cuál es el aumento de velocidad.

Tenemos los mismos datos que en la implementación en serie, una imagen hiperespectral 715x1096 con 102 bandas cada píxel.

Implementación

Lo primero que tenemos que identificar es qué información tienen que intercambiar los procesos y cuándo. Recordando el algoritmo sabemos que, en él, hay una imagen que solo se va a leer sobre la

que vamos a realizar los cálculos dando como resultado los centroides que utiliza el algoritmo para agrupar y a qué centroide pertenece cada píxel hiperespectral.

Como la imagen solo la van a leer, los datos no van a cambiar por lo que con que se envíe una vez al principio del algoritmo sería suficiente, para paralelizar el algoritmo repartimos esta imagen equitativamente entre todos los procesos. Los centroides cambian en cada iteración y son necesarios para poder realizar los cálculos, por lo que en cada iteración se comparten los centroides actualizados. Las etiquetas de cada píxel también van a cambiar a lo largo de cada iteración, pero como lo que interesa de ellas es el resultado final solo se necesitan juntar los resultados al final de la ejecución del algoritmo.

Previo a la ejecución del algoritmo hay dos tareas que no se deben de duplicar, son la lectura de la imagen y la inicialización de centroides, esto quiere decir que solo lo debe realizar un proceso y en MPI se utiliza lo que llamamos el proceso “MASTER”, este proceso es igual que cualquier otro, todos los procesos tienen un identificador y lo más común es que el MASTER tenga de identificador el número 0, de esta forma podemos indicar que según el id del proceso realice una tarea o no, es el caso de la lectura de la imagen y la inicialización de los centroides que solo lo va a hacer el proceso MASTER.

Todas las comunicaciones se hacen mediante lo que en MPI se denominan comunicadores, los comunicadores son identificadores de canales de comunicación virtuales, ya que no son conexiones reales físicas, de forma que cuando un proceso hace una llamada a MPI ya sea de envío o de recepción, todos los procesos que estén a otro lado de la llamada y que tengan el mismo comunicador indicado van a realizarla. Comúnmente se utiliza el comunicador “MPI_COMM_WORLD” o por su traducción comunicador del mundo de mpi, este comunicador indica que todos los procesos intervienen en esa comunicación.

En nuestra implementación, tras estimar qué valores podían ser buenos, hemos establecido 5 centroides, 50 iteraciones máximo para el bucle principal y una convergencia de 0,2.

Repartir la imagen

MPI nos proporciona una función con la que podemos enviar un array a varios procesos, pero sin necesidad de tener que enviarla entera, pudiendo recibir cada proceso solo una porción del array que se pretende repartir sin que el receptor tenga que preocuparse de controlar la información recibida. Este método es "MPI_Scatter", todos los procesos la ejecutan con los parámetros en este orden; array a repartir, cantidad de datos que tiene que recibir cada proceso, tipo de dato, array en el que recibir, cantidad de datos a recibir, tipo de dato, proceso encargado del envío y el canal de comunicación por el que se realiza. Como en la llamada se indica qué proceso se va a encargar del envío, los receptores solo se tienen que preocupar del array en el que van a recibir y la cantidad.

```
//repartir la imagen
MPI_Scatter(data, data_per_node, MPI_FLOAT, data_part, data_per_node, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
```

Tras esta llamada ya todos los procesos tendrán la porción de la imagen que les corresponde y se ejecutará únicamente una vez antes del bucle principal del algoritmo.

Repartir las etiquetas

El segundo paso antes del inicio del bucle principal es repartir el array donde se van a almacenar las etiquetas con las que se van a identificar al clúster que pertenezca cada clúster. Como cada proceso tendrá una porción proporcionada usamos el mismo método con el que se ha repartido la imagen.

```
//repartir las etiquetas
MPI_Scatter(labels, labels_per_node, MPI_INT, labels_part, labels_per_node, MPI_INT, MASTER, MPI_COMM_WORLD);
```

Estas etiquetas son las que nos servirán al final para mostrar el resultado de la imagen según las agrupaciones que se formen en la ejecución del algoritmo, gracias al método ”scatter-gather” más tarde podremos reunir estos datos de vuelta de una forma sencilla.

Compartir los centroides

Los centroides no se van a repartir, sino que compartir, cada proceso va a procesar un todo de la imagen, pero independientemente del trozo que le corresponda va a necesitar todos los centroides para poder determinar a qué agrupación corresponde cada uno de los píxeles de su porción.

Los centroides son inicializados en el proceso MASTER y este se tiene que encargar de compartirlo con el resto de los procesos. Para esto se puede utilizar el paradigma de “broadcast” que MPI permite con la llamada MPI_Bcast, en ella, los procesos indican el puntero donde se debe de almacenar, la cantidad de los datos, el tipo de datos, el id del proceso que se va a encargar de enviar al resto de procesos una copia de su puntero y el comunicador por el que se va a realizar el envío.

```
//se copian los centroides en todos los procesos
MPI_Bcast(centers, centers_size, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
```

Tras la llamada, todos los procesos van a tener en el puntero “centers” almacenado una copia igual de los centroides.

Ejecución serial de todos los procesos

Como en la ejecución en serie, una vez todos los procesos tienen los datos necesarios, realizan los cálculos principales del algoritmo, la asignación de cada punto a su clúster correspondiente y actualizar las posiciones de los centroides.

Como el cálculo no varía se explica brevemente indicando los pequeños cambios debidos a la ejecución paralela:

Para asignar los puntos a un clúster (point to cluster = ptc) se recorren todos los puntos que tiene el proceso y se calcula la distancia a cada centroide, después, se guarda en el array de etiquetas el identificador del clúster con la menor distancia al punto.

```
//assignPointsToClusters
for(int ptc = 0; ptc < pixels_per_node; ptc++){
    minDist = FLOAT_MAX;
    minLabel = -1;
    for(int cs = 0; cs < K; cs++){
        dist = euclideanDistance(data_part, centers, ptc, cs);
        if(dist < minDist){
            minDist = dist;
            minLabel = cs;
        }
    }
    labels_part[ptc] = minLabel;
}
```

Para actualizar los centroides necesitamos un contador para la cantidad de puntos agrupados en cada centroide y un nuevo array donde vamos a calcular los nuevos centroides, recorremos las etiquetas para ir aumentando los contadores de cada clúster y hacemos el sumatorio de las bandas de los píxeles que pertenecen a ese clúster.

```
//updateCenters
int pixel_offset, clusterCounts[K], label;
for(int ks = 0; ks < K; ks++){
    clusterCounts[ks] = 0;
}
for(int px = 0; px < labels_per_node; px++){
    pixel_offset = px * BANDS;
    label = labels_part[px];
    clusterCounts[label]++;
    for(int bs = 0; bs < BANDS; bs++){
        new_centers[(label * BANDS) + bs] += data_part[pixel_offset + bs];
    }
}
```

Creación de nuevos centroides

Al crear los nuevos centroides surge una característica debido a la paralelización, todos los procesos calcularán un nuevo set de centroides, pero luego esos tienen que ser comunes, por lo que debemos determinar, según la proporción de trabajo de cada proceso, cuánto afectará al estado global.

Para ello, ponderamos el cálculo del nuevo centroide según el trabajo realizado, esta ponderación será de entre 0 y 1 y se calcula dividiendo la cantidad de puntos procesados respecto del total de la imagen, con esta ponderación multiplicamos por el valor de las bandas de los centroides para cuando después se pongan en común (se sumen) cada una valga la proporción que le corresponde.

```
int center_offset;
for(int ks = 0; ks < K; ks++){
    center_offset = ks * BANDS;
    if(clusterCounts[ks] > 1){
        for(int bs = 0; bs < BANDS; bs++){
            new_centers[center_offset + bs] = new_centers[center_offset + bs] / (float)clusterCounts[ks];
            new_centers[center_offset + bs] = new_centers[center_offset + bs] * ponder;
        }
    }
}
```

Recoger resultados y ver convergencia

Ya todos los procesos han calculado sus centroides, el MASTER tiene que recoger los nuevos centroides para juntarlos y comprobar si el algoritmo ha convergido y debe terminar. Para recoger los centroides de todos los procesos se utiliza el método MPI_Gather, con este método un proceso puede recoger distintos datos de varios procesos y los procesos solo se tienen que encargar de llamar al método sin preocuparse del envío. Para el emisor son importantes los tres primeros parámetros del método que son el puntero de los datos a enviar, su tamaño y su tipo, después hay 3 parámetros que utiliza el receptor que coinciden con los del emisor con el cambio de que la cantidad de datos a recibir indica los datos que va a recibir de cada proceso, pero debe tener en cuenta que en el puntero donde los va a almacenar tiene que haber suficiente espacio, y finalmente el comunicador.

```
//se recogen las medias ponderadas de la actualización de centroides teniendo en cuenta si todos los procesos tienen
MPI_Gather(new_centers, centers_size, MPI_FLOAT, all_new_centers, centers_size, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
```

Ahora, el MASTER tiene todos los centroides ponderados por los procesos, pero recibidos en distintos espacios de memoria y que deben unirse en un array definitivo con los nuevos centroides calculados. Usando el array donde se habían calculado las ponderaciones de los centroides, que vaciamos su contenido, y hacemos el sumatorio de cada punto de los diferentes arrays de centroides.

```
//juntamos todos los nuevos centroides ponderados en el master
if(rank == MASTER){
    for(int ac = 0; ac < (K * BANDS); ac++){
        new_centers[ac] = 0;
    }
    for(int nodes = 0; nodes < size; nodes++){
        for(int cs = 0; cs < K; cs++){
            center_offset = cs * BANDS;
            for(int bs = 0; bs < BANDS; bs++){
                new_centers[center_offset + bs] += all_new_centers[(nodes * centers_size) + center_offset + bs];
            }
        }
    }
```

Para ver si converge, calculamos la distancia euclídea entre los nuevos centroides y los que ya teníamos y si la mayor distancia de cualquier centroide respecto de su posición en la iteración anterior es menor que la convergencia que tenemos determinada, establecemos a “TRUE” una variable que indica al bucle de la ejecución principal si parar. También copiamos los nuevos centroides en el array de centroides para volver a enviarlos a todos los procesos y que en la siguiente iteración se pueda volver a calcular la convergencia. Por último, incrementamos la variable que guarda el número de iteraciones que lleva la ejecución puesto que también hay un límite de iteraciones que pueden realizarse.

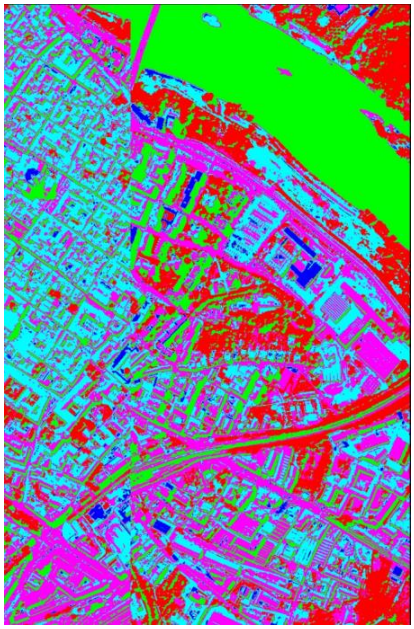
```
    //miramos si converge
    maxMovement = FLOAT_MIN;
    for(int ks = 0; ks < K; ks++){
        dist = euclideanDistance(new_centers, centers, ks, ks);
        if(dist > maxMovement)
            maxMovement = dist;
    }
    if(maxMovement < CONVERGENCE)
        converged = TRUE;

    memcpy(centers, new_centers, centers_size * sizeof(float));
}

MPI_Bcast(&converged, 1, MPI_FLOAT, MASTER, MPI_COMM_WORLD);

iter++;
```

Cuando se dan las condiciones para terminar el bucle principal, solo queda que el MASTER recoja las etiquetas finales de cada píxel hiperespectral con las que se van a crear la imagen y finalizar la ejecución de MPI.

```
MPI_Gather(labels_part, labels_per_node, MPI_INT, labels, labels_per_node, MPI_INT, MASTER, MPI_COMM_WORLD);
MPI_Finalize();
```

Resultado de la ejecución

Con la librería de opencv creamos una imagen asignando a cada etiqueta un color diferente. La imagen resultante queda así para n=2:



A continuación, se muestra la tabla de tiempos para el tiempo de ejecución con la implementación de kmeans del hito 1 junto con la solución que incorpora openmp para distintos threads (n). Se han elegido los siguientes hiper-parámetros:
- k = 5.
- Epsilon = 0.2.
- Máximo de iteraciones = 50.

Las mediciones se han hecho en un ordenador portátil Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, 8GB de memoria RAM y cache de 3 MiB.

La ejecución en serie en este portátil con estas características duraba alrededor de 4 minutos, tras la paralelización con MPI y haber probado varias ejecuciones el tiempo se ha disminuido a un intervalo entre 38 y 45 segundos.

Este tiempo es mucho mejor comparado con los cerca de 4 minutos que tarda en su versión en serie lo que supone un speedup de 6.

OpenMP

Introducción

Después de la implementación de k-means en serie y con MPI (paso de mensajes) ahora vamos a implementarlo con el método de memoria compartida, para ello nos

aprovechamos de “pragmas” que indicamos en el código para que el compilador se encargue del proceso de usar la memoria compartida.

Respecto a las características de la ejecución son las mismas que en las anteriores implementaciones, una imagen de 1096x715x102, con 5 centroides, un máximo de 50 iteraciones y una convergencia de 0,2.

Implementación

Inicio

Lo único que se puede destacar en la preparación antes del bucle principal es poder obtener el número de hilos que va a usar omp.

```
int num_threads = omp_get_num_threads();
```

Bucle principal

A diferencia de MPI, ya con OpenMP podemos estructurar nuestro código con funciones, de forma que el bucle principal se ve la idea principal de la ejecución y podemos ver en los diferentes métodos el uso de OpenMP.

```
bool converged = false;
int iter = 0;
while(!converged && iter < MAX_ITER){
    assignPointsToClusters(data, centers, labels);
    updateCenters(labels, new_centers, data);
    if(check_convergence(centers, new_centers) <= CONVERGENCE)
        converged = true;
    else
        memcpy(centers, new_centers, centers_size * sizeof(float));
    iter++;
}
```

Asignar puntos a los centroides

Primero hay que entender cómo utilizar la memoria OpenMP:

```
void assignPointsToClusters(float *data, float *centers, int *labels){
    float minDist, minLabel, dist;
    int omp_num_threads = omp_get_num_threads(), thread_id;
    size_t chunk_size = data_size / omp_num_threads;

    #pragma omp parallel private(minDist, minLabel, dist, thread_id)
    thread_id = omp_get_thread_num();
    {
```

Aquí hay diferentes formas de tener variables en el código; todas las variables que son creadas previas al #pragma son por defecto compartidas, por tanto, si quieres que los diferentes hilos no las compartan dentro de la llamada del #pragma tienes que

indicarlo con ”private()”, en este caso, como las variables que vamos a usar para el cálculo de las distancias tienen que ser privadas por hilo las indicamos, también el id del hilo, sin embargo, ”chunk_size”, ”data” y ”centers” solo la van a leer por lo que no es necesario privatizarla, en el caso de ”labels” sí que van a hacer escrituras todos en ella pero como cada hilo va a escribir en diferentes posiciones del array no hay problema en el uso común.

En este caso, el pragma que se utiliza es “parallel”, lo que hace este pragma es crear la bifurcación en la línea de ejecución, es decir, entre las llaves se crean los distintos hilos que van a ejecutar el código que contienen y cuando se llega a la llave de cierre todos los hilos se matan, todo esto sin la preocupación por parte del programador y sin necesidad de preocuparse por el manejo de la memoria compartida.

```
#pragma omp parallel private(minDist, minLabel, dist, thread_id)
thread_id = omp_get_thread_num();
{
    //varios hilos hacen el for con 3 variables privadas repartiendo bloques de los datos
    #pragma omp parallel for schedule (static, chunk_size)
    for(int point_to_cluster = 0; point_to_cluster < n_pixels; point_to_cluster++){
        minDist = FLOAT_MAX;
        minLabel = -1;
        for(int center = 0; center < K; center++){
            dist = euclideanDistance(data, centers, point_to_cluster, center);
            if(dist < minDist){
                minDist = dist;
                minLabel = center;
            }
        }
        labels[point_to_cluster] = minLabel;
    }
}
```

Una vez creados los hilos comienza la ejecución, el siguiente punto importante es el pragma que hay dentro del parallel, en este caso es un ”for”. El funcionamiento de este consiste en paralelizar el bucle for, aunque el pragma anterior ha creado distintos hilos de ejecución, todos van a ejecutar el código completo que está entre las llaves, lo que supone que todos ejecutarían el for completo, de la forma en la que vemos en la imagen, dentro de la paralelización entre hilos vamos a dividir el for entre los diferentes hilos para que cada uno haga una parte del bucle.

En el pragma del for tenemos también la directiva “schedule”, con ella indicamos la forma en la que se va a repartir el bucle, “static” quiere decir que los trozos en los que se va a dividir el bucle se van a repartir en orden circular, y el segundo parámetro es el tamaño que va a tener cada trozo en el que se divida el bucle. El resultado de esto es que el bucle se va a dividir en bloques de tamaño chunk_size y se van a repartir de forma circular.

Al dividir el bucle en bloques y repartirlo las escrituras en el array labels se harán de forma correcta puesto que el índice donde se escribe solo va a aparecer una vez entre todas las ejecuciones de los distintos hilos.

Actualizar los centroides

En esta función comenzamos de la misma manera, declaramos las variables que vamos a usar después indicando en el pragma de la paralelización cuáles van a ser privadas. Es importante destacar en este caso que el array clusterCounts es público, pero por motivos de rendimiento que vamos a ver ahora después creamos otro igual llamado clusterCounts_thread que va a ser privado.

El primer paso en la actualización de los centroides es incrementar los contadores de etiquetas que vea cada hilo y hacer los sumatorios de cada punto de la imagen hiperespectral, OpenMP tiene una directiva para utilizar algoritmos de reducción que son muy eficientes para realizar de forma muy eficiente operaciones aritméticas sobre un dato, en este caso la hemos podido utilizar para hacer los sumatorios de cada punto.

Aquí es donde está la importancia de la variable privada nombrada antes, como en el incremento contando cuántos puntos pertenecen a cada clúster es muy probable que varios hilos intenten escribir en la misma posición del array, entonces, hemos hecho copias de ese array para cada hilo y después poder sumar los resultados. Aun así, al sumar tenemos el mismo problema, pero podemos usar el pragma de OpenMP "critical" que sirve sección crítica que solo puede ser ejecutada por un hilo a la vez, así, podemos paralelizar gran parte del contador de clusters dejando fuera la suma final, que por su tamaño la sobrecarga de paralelizarlo lo supera.

Por último, calculamos el valor de los nuevos centroides. Como para hacer este cálculo necesitamos tener el contador con los valores finales, por lo que usamos una barrera de OpenMP que para a todos los hilos en ese punto de ejecución hasta que todos estén en ese punto. Ya, se paraleliza el bucle for para hacer el cálculo y el resultado son los centroides actualizados.

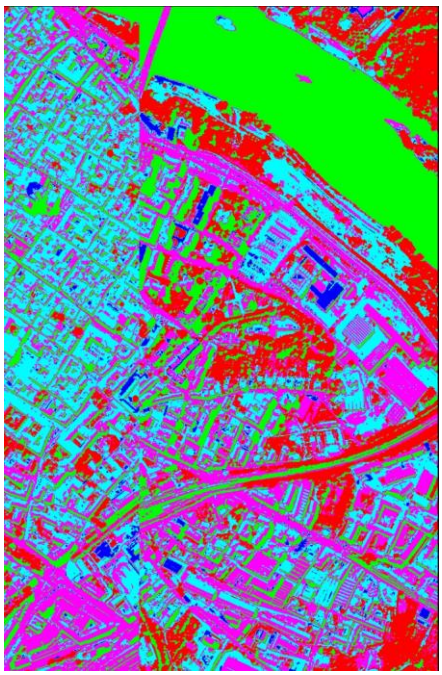
Comprobar la convergencia

Finalmente, comprobamos si el bucle principal debe acabar.

Este bucle solo tiene 5 iteraciones y podría plantearse la duda de si la sobrecarga comparada con la aceleración pueda merecer la pena, hemos considerado que sí porque dentro del cálculo de la distancia euclídea contiene un bucle. También hay que añadir que como la variable que guarda la mayor distancia puede provocar problemas a la hora de que diferentes hilos escriban en ella así que la protegemos como una operación atómica, que solo la puede realizar un hilo a la vez.

Resultado de la ejecución

Con la librería de opencv creamos una imagen asignando a cada etiqueta un color diferente. La imagen resultante queda así para n=2:



A continuación, se muestra la tabla de tiempos para el tiempo de ejecución con la implementación de kmeans del hito 1 junto con la solución que incorpora openmp para distintos threads (n). Se han elegido los siguientes hiper-parámetros:
- k = 5.
- Epsilon = 0.2.
- Máximo de iteraciones = 50.

Las mediciones se han hecho en un ordenador portátil Intel(R) Core(TM) i5-6200 CPU @ 2.30GHz, 8GB de memoria RAM y caché de 3 MiB.

Respecto al tiempo de ejecución es más variable que su homólogo en mpi, hay que tener en cuenta que al inicializarse aleatoriamente los centroides esto afecte a la cantidad de iteraciones y por ende al tiempo de ejecución, el tiempo mínimo experimentado ha sido de 38 segundos mientras que el máximo 120, más o menos la media de tiempo de ejecución está en el intervalo de 60-70 segundos.

Comparando con los tiempos anteriores tenemos una clara mejora respecto a los 4 minutos que podía tardar en la ejecución en serie puesto que, como mínimo, según lo experimentado tenemos el doble de velocidad en la ejecución.

Mientras que comparando con MPI vemos que tarda más puesto que los tiempos que observábamos eran entre 38-45 segundos, esto se puede deber a la forma en que está diseñada la arquitectura de la memoria que penalice para este tipo de ejecuciones en comparación con el paso de mensajes.