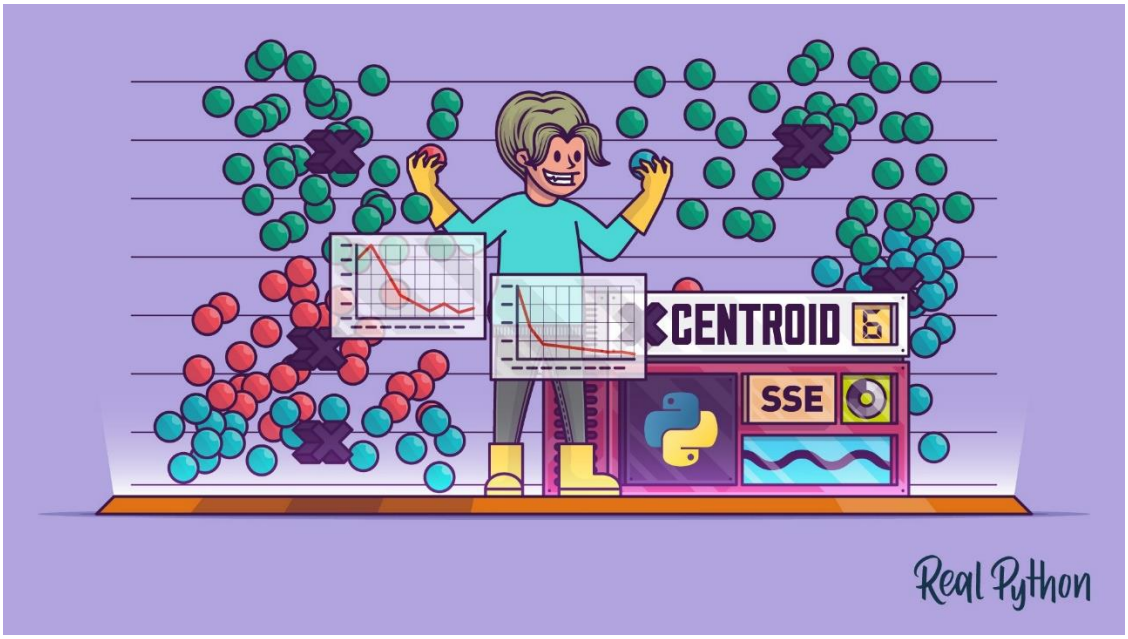
Escuela Superior de Informática

Máster Universitario en Ingeniería Informática

REDUCCIÓN DEL TIEMPO DE CÓMPUTO EN ALGORITMOS DE CLUSTERING

LABORATORIO – TRABAJO FINAL

COMPUTACIÓN DE ALTAS PRESTACIONES



Alberto Castellanos Gallego

8 de junio de 2022

CONTEXTO

Este trabajo tiene como objetivo conseguir desarrollar una versión del algoritmo de clustering KMeans paralelizada y, por tanto, intentar obtener una reducción del tiempo de ejecución con la ayuda de, en mi caso, OpenMP. El desarrollo parte de una versión secuencial a la que, con un estudio previo, se han aplicado directivas y estrategias de paralelización para alcanzar el objetivo. Todo ello se explicará en siguientes apartados.

PROBLEMA

Es en este apartado donde se irá indicando cada uno de los pasos seguidos junto con los problemas que fueron surgiendo en el estudio y desarrollo.

ESTUDIO PREVIO.

Primero que nada, la cuestión estaba entre si elegir como lenguaje para la paralelización OpenMP o MPI. Me decanté por OpenMP ya que, aparte de que su sintaxis me era más fácil de entender, su uso de memoria compartida me iba a facilitar la tarea y dar mejores resultados de rendimiento, pues, a diferencia de MPI, no necesita una comunicación tan exhaustiva entre las tareas teniendo en cuenta que mi volumen de datos iba a ser más bien alto y complejo (3 dimensiones).

En cuanto al lenguaje de programación, barajé las opciones de C y Python. En un principio opté por Python, pues tengo mejor desenvoltura. Sin embargo, tuve que desechar esta opción por el hecho de que, después de tener el código secuencial desarrollado, me di cuenta de la dificultad de aplicar OpenMP a Python, puesto que requería de la utilización de Cython que mezclaba Python con C y además era necesario forzar el interpretador global de Python porque, por defecto, este muchas veces forzaba a realizar una operación en un único hilo.

Tras los resultados anteriores, desarrollé el algoritmo en lenguaje C en el que me llevó más tiempo, pero finalmente no me dio tantos problemas, únicamente la gestión de asignación de memoria con las directivas malloc y calloc.

ESTRATEGIA SEGUIDA Y DESCOMPOSICIÓN EN TAREAS

El primer paso a la hora de desarrollar un algoritmo paralelo es descomponer el problema en tareas que se pueden ejecutar de forma concurrente.

Los pasos que se siguen en el algoritmo son:

```
#PASO1: COGEMOS ALEATORIAMENTE PUNTOS RANDOM COMO NUESTROS CENTROIDES
#PASO2: CALCULAMOS LA DISTANCIA EUCLIDEA ENTRE CADA DATO Y CADA CENTROIDE
#PASO3: ASIGNAMOS CADA DATO A SU CENTROIDE MÁS CERCANO
#PASO4: ACTUALIZAMOS LA POSICIÓN DE LOS CENTROIDES TENIENDO EN CUENTA LA DISTANCIA MEDIA DE LOS
DATOS DE CADA CLUSTER
```

En mi caso, los pasos que he paralelizado han sido del 2 al 4. Y dentro de cada uno de ellos, si así ha sido necesario, he tenido que restringir el acceso a memoria de las variables.

La estrategia que he seguido ha sido, en primer lugar, trocear los datos de entrada según los hilos que haya indicado por terminal, es decir, he seguido una descomposición basada en datos. Estos datos están compuestos por coordenadas de 3 dimensiones en cada una de las N filas totales del archivo .txt. Como he dicho, se repartió a cada hilo un trozo de los datos de entrada como se ve en la siguiente imagen.

```
int longitudHilo = N / nHilos;
int inicio = hiloID * longitudHilo;
int fin = inicio + longitudHilo;

if (fin > N)
{
  fin = N;
  longitudHilo = inicio - fin;
}
```

Después de realizar toda la lógica del algoritmo y en cada iteración, cada hilo, de uno en uno, actualiza los valores de los nuevos centroides hasta obtener la convergencia deseada.

```
#pragma omp critical
  {
    for (int i = 0; i < K; i++) {
      if (contadorCluster[i] == 0) {
        // aquí sólo entra si el cluster no tiene puntos
        continue;
      }

      //Actualizamos los centroides
      centroidesGlobal[((contIteracion + 1) * K + i) * 3] = centroideActual[(i * 3)] / (float)contadorCluster[i];
      centroidesGlobal[((contIteracion + 1) * K + i) * 3 + 1] = centroideActual[(i * 3) + 1] / (float)contadorCluster[i];
      centroidesGlobal[((contIteracion + 1) * K + i) * 3 + 2] = centroideActual[(i * 3) + 2] / (float)contadorCluster[i];
    }
  }
```

```
//Actualizamos la asignación de puntos a sus respectivos clusters
for (int i = inicio; i < fin; i++) {
  puntosClusterGlobal[i * 4] = puntos[i * 3];
  puntosClusterGlobal[i * 4 + 1] = puntos[i * 3 + 1];
  puntosClusterGlobal[i * 4 + 2] = puntos[i * 3 + 2];
  puntosClusterGlobal[i * 4 + 3] = (float)idClusterActual[i - inicio];
}
```

COMUNICACIÓN Y DEPENDENCIA ENTRE DATOS

Como he comentado al principio de este informe, seleccioné OpenMP como lenguaje para la paralelización. Esto implica que la comunicación entre las tareas sea a través de memoria

compartida en vez de por paso de mensajes (MPI), lo que ayuda a que no haya sobrecarga en la comunicación entre tareas, pero quizá sí que pueda darse el caso de sufrir Waiting For Contention, donde varios procesos pueden querer acceder al mismo espacio de memoria, aunque para mí, preferible esta sobrecarga a la que pudiese haber obtenido por la comunicación en el paso de mensajes.

El tamaño de cada tarea será en todos los casos la misma, salvo para la que obtenga el último bloque de datos disponible, que será menos.

MAPEO DE TAREAS

Una vez el problema ha sido descompuesto en tareas concurrentes, estas tienen que ser mapeadas a cada proceso con el objetivo de minimizar sobrecargas.

La técnica de mapeo que he seguido ha sido la estática porque yo, en un principio, se el tamaño de los datos de entrada y puedo minimizar prácticamente al mínimo los posibles problemas de balanceo de carga, ya que, únicamente tendrá menos carga computacional aquella tarea que obtenga el último bloque de datos.

RESULTADOS

Intel(R) Core (TM) i7-6700HQ CPU @ 2.60GHz de 4 núcleos y 8 hilos.

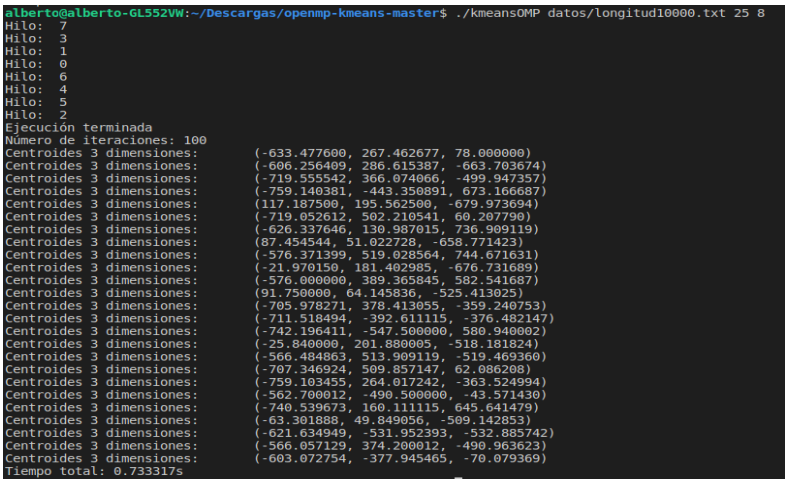
Algoritmo Secuencial

Clústers ->	5	10	25
5000 datos	0.0682(s)	0.1687(s)	0.4191(s)
10000 datos	0.1231(s)	0.6637(s)	1.3900(s)

```
alberto@alberto-GL552VW:~/Descargas/openmp-kmeans-master$ ./kmeans datos/longitud10000.txt 25
Ejecución terminada
Número de iteraciones: 65
Centroides 3 dimensiones:       (510.688477, 743.380615, 50.454956)
Centroides 3 dimensiones:       (197.639633, -662.666687, -571.240906)
Centroides 3 dimensiones:       (376.943909, 577.040466, 16.795506)
Centroides 3 dimensiones:       (43.748314, -527.563416, -714.094421)
Centroides 3 dimensiones:       (375.117981, -357.990936, -691.000000)
Centroides 3 dimensiones:       (-390.321991, -402.065796, -749.717102)
Centroides 3 dimensiones:       (-541.664490, 608.219055, -50.938145)
Centroides 3 dimensiones:       (194.729385, -517.949829, -556.713257)
Centroides 3 dimensiones:       (512.638000, -523.710938, -712.489685)
Centroides 3 dimensiones:       (519.884949, 34.820183, -757.812866)
Centroides 3 dimensiones:       (284.910095, -364.179688, -581.046875)
Centroides 3 dimensiones:       (-508.817719, -6.489130, -747.036255)
Centroides 3 dimensiones:       (138.668472, 729.035400, -52.273586)
Centroides 3 dimensiones:       (71.806602, -510.542267, -625.241760)
Centroides 3 dimensiones:       (-533.298096, -521.835388, -576.444092)
Centroides 3 dimensiones:       (374.316772, -495.219635, -596.633057)
Centroides 3 dimensiones:       (-386.948334, -68.530388, -625.062622)
Centroides 3 dimensiones:       (241.836090, 85.535805, -617.372864)
Centroides 3 dimensiones:       (280.972839, -663.381531, -717.547668)
Centroides 3 dimensiones:       (375.076935, -504.254150, -753.235168)
Centroides 3 dimensiones:       (-423.045135, 590.958862, 109.997429)
Centroides 3 dimensiones:       (175.652954, -668.846130, -722.505493)
Centroides 3 dimensiones:       (513.054932, 71.019913, -606.752197)
Centroides 3 dimensiones:       (135.188049, 716.703613, 112.079521)
Centroides 3 dimensiones:       (49.436146, -659.043457, -578.903748)
Tiempo total: 1.332272s
```

Algoritmo Paralelo

Clústers ->	10	15	25
5000 datos	0.3792(s)	0.5154(s)	0.6444(s)
10000 datos	0.2962(s)	0.5923(s)	0.7333(s)



Las conclusiones extraídas de las pruebas hacen ver una correcta correlación en los tiempos obtenidos en cada una de las pruebas, obteniendo bastante ganancia en el caso de la ejecución con OpenMP.
Cabe destacar que, al hacer pruebas elevando los hilos, este se hacía más lento que el secuencial. Pienso que puede ser debido a que elevé muchísimo la granularidad, es decir, descompuse el problema en muchas tareas (Granularidad fina), lo que provocó una posible sobrecarga en la comunicación más que en la carga computacional. Este tipo de granularidad me reduce las posibilidades de obtener speedup, como pude comprobar.

Soy consciente de que 25 clústers para la cantidad de datos que he manejado es muchísimo, pero simplemente lo que he probado así para que los tiempos obtenidos sean más vistosos

CÓDIGO FUENTE

El código fuente se encuentra en el repositorio:
https://github.com/Alberto811/TrabajoFinalCAP.git.
Está incluido un archivo makefile con el que ejecutaremos:

Para el código secuencial: ***make secuencial***
Para el código paralelo: ***make paralelo***

Como datos de salida obtendremos algo parecido a lo mostrado en las imágenes anteriores. Tantas filas de coordenadas de los centroides como clústers se hayan especificado, en mi caso 25.