COPUTACIÓN DE ALTAS PRESTACIONES

TRABAJO FINAL

Richard Alexander Jiménez Rijo

INTRODUCCIÓN

Se nos asigno la reducción del tiempo de ejecución de los algoritmos de clustering en entornos de gran volumen de datos utilizando técnicas de paralelización y lenguajes que den soporte a las mismas como son MPI y/o OpenMP.

DESARROLLO DEL PROYECTO

Se ha seleccionado un algoritmo de kmeans simple en C++ encontrado en una página web para llevar a cabo el procedimiento. Dicho código puede encontrarse en el siguiente enlace: [Algoritmo de agrupación en clústeres K-means código en lenguaje C - programador clic (programmerclick.com)]

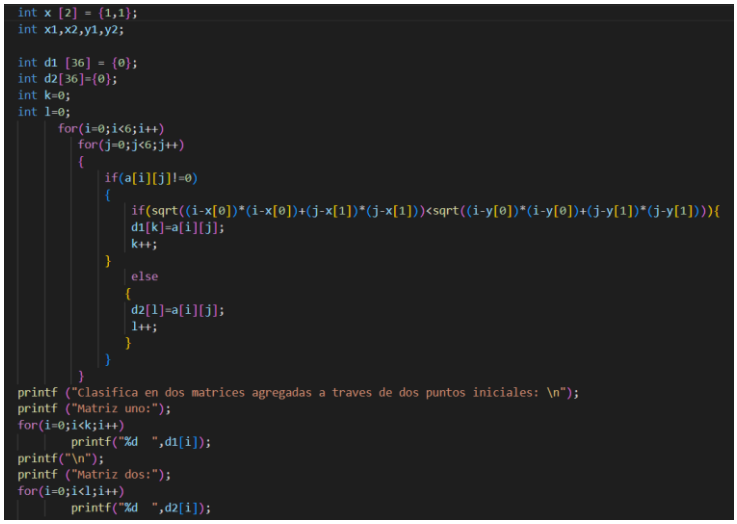
Las pruebas se llevarán a cabo en un equipo que cuenta con 8 núcleos y 16 procesadores lógicos. Las tareas de nuestro código se organizan en tres fases, la iniciación, la agrupación y la actualización de centroides.

La iniciación:

```
int i,j;
int a[6][6];
srand (time (0));

   for(i=0;i<6;i++)
   {
    for(j=0;j<6;j++)

    a [i] [j] = 0 + rand ()% 2;
    }

   for(i=0;i<6;i++)
    for(j=0;j<6;j++)
   {
      if(a[i][j]==1)
    a[i][j]=i*6+j+1;
    }

    printf ("Generar puntos de objeto aleatoriamente: \n");
    for(i=0;i<6;i++)
    for(j=0;j<6;j++)
    {
     if(j!=5)
     printf(" %02d  ",a[i][j]);
     else
    printf(" %02d  \n",a[i][j]);
    }
    printf("\n");printf("\n");
```

Es la primera parte del código del algoritmo, aquí es donde se inician las variables y se les asigna los valores a trabajar. Se elige aleatoriamente k objetos como los centroides iniciales para k grupos.

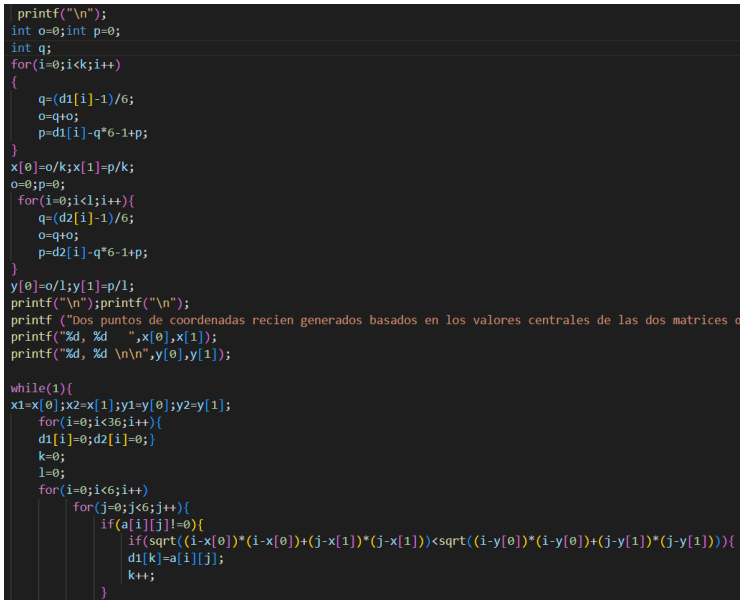
La agrupación:



```
int x [2] = {1,1};
int x1,x2,y1,y2;

int d1 [36] = {0};
int d2[36]={0};
int k=0;
int l=0;
      for(i=0;i<6;i++)
         for(j=0;j<6;j++)
         {
             if(a[i][j]!=0)
             {
                 if(sqrt((i-x[0])*(i-x[0])+(j-x[1])*(j-x[1]))<sqrt((i-y[0])*(i-y[0])+(j-y[1])*(j-y[1]))){
                 d1[k]=a[i][j];
                 k++;
             }
                 else
                {
                 d2[l]=a[i][j];
                 l++;
                }
             }
         }
printf ("Clasifica en dos matrices agregadas a traves de dos puntos iniciales: \n");
printf ("Matriz uno:");
for(i=0;i<k;i++)
        printf("%d  ",d1[i]);
printf("\n");
printf ("Matriz dos:");
for(i=0;i<l;i++)
        printf("%d  ",d2[i]);
```

Aquí se calcula la distancia entre los objetos y los centroides. Nos identifica que tan alejados pueden estar los objetos de los centroides y cuales se encuentran más cerca de estos.

La actualización de centroides:



```
 printf("\n");
int o=0;int p=0;
int q;
for(i=0;i<k;i++)
{
    q=(d1[i]-1)/6;
    o=q+o;
    p=d1[i]-q*6-1+p;
}
x[0]=o/k;x[1]=p/k;
o=0;p=0;
 for(i=0;i<l;i++){
    q=(d2[i]-1)/6;
    o=q+o;
    p=d2[i]-q*6-1+p;
}
y[0]=o/l;y[1]=p/l;
printf("\n");printf("\n");
printf ("Dos puntos de coordenadas recien generados basados en los valores centrales de las dos matrices
printf("%d, %d   ",x[0],x[1]);
printf("%d, %d \n\n",y[0],y[1]);

while(1){
x1=x[0];x2=x[1];y1=y[0];y2=y[1];
    for(i=0;i<36;i++){
    d1[i]=0;d2[i]=0;}
    k=0;
    l=0;
    for(i=0;i<6;i++)
        for(j=0;j<6;j++){
            if(a[i][j]!=0){
                if(sqrt((i-x[0])*(i-x[0])+(j-x[1])*(j-x[1]))<sqrt((i-y[0])*(i-y[0])+(j-y[1])*(j-y[1]))){
                d1[k]=a[i][j];
                k++;
            }
```

Aquí obtenemos los objetos más cerca de los centroides en base a puntos de coordenadas y se revisa si ha habido algún cambio en los centroides, esto no se ven tan afectados debido a que estamos trabajando con un ejemplo que genera sus propios objetos y no con una base de datos real.

TIEMPOS DE EJECUCIÓN

Se realizaron los tiempos de ejecución con las librerías de **time.h** y **unistd.h** para obtener los tiempos de cada de una de las fases del algoritmo y del algoritmo completo en sí.

#include <time.h>: es una librería que añade funciones para manipular la fecha y la hora.

#include <unistd.h>: es una Liberia que contiene funciones de contenedor de las llamadas el sistema básico del sistema operativo.

Lo que hice fue almacenar el tiempo de ejecución teniendo en cuenta el momento en el se inicia menos el momento en el que termina. El código que utilizados para medir los tiempos de ejecución fue el siguiente:

```c
#include <stdio.h>
#include <time.h>       // for clock_t, clock(), CLOCKS_PER_SEC
#include <unistd.h>     // for sleep()

// función principal para encontrar el tiempo de ejecución de un programa en C
int main()
{
    // para almacenar el tiempo de ejecución del código
    double time_spent = 0.0;

    clock_t begin = clock();

    // hacer algunas cosas aquí
    sleep(3);

    clock_t end = clock();

    // calcula el tiempo transcurrido encontrando la diferencia (end - begin) y
    // dividiendo la diferencia por CLOCKS_PER_SEC para convertir a segundos
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("The elapsed time is %f seconds", time_spent);

    return 0;
}
```

En base a este código y aplicando en el algoritmo ya seleccionado se pudieron obtener los tiempos de ejecución aproximados que se tarda el ejemplo de k means en sus diferentes fases y en su ejecución completa.

Iniciación: el tiempo de ejecución es de 3.006000 segundos.

Agrupación: el tiempo de ejecución es de 3.005000 segundos.

Actualización de centroides: el tiempo total de ejecución del algoritmo es de 3.034000 segundos.

PARALELIZACIÓN

Para la paralelización me voy a centrar en todos los bucles que se encuentran en nuestro algoritmo y voy a utilizar openMP para la mejora de estos e intentar un conseguir un tiempo de ejecución más rápido entre cada una de las fases del script. Microsoft nos brinda información de como podemos utilizar openMP para paralelizar nuestros algoritmos:

Estaremos usando la directiva parallel for, aquí la variable de iteración del bucle es privada de forma predeterminada, por lo que no es necesario especificarla explícitamente en una cláusula privada.

```C++
#pragma omp parallel for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
```

También vamos a probar con la cláusula nowait, como tenemos muchos bucles en nuestro algoritmo para evitar la barrera que aparece al final de los for.

```C++
#pragma omp parallel
{
    #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
}
```

Probare cada uno de los dos métodos de paralelización que seleccione de openMP en cada una de las fases, así como en su ejecución final para proceder a ver si se puede notar algun cambio significativo entre las ejecuciones. También vamos a utilizar el código **#pragma omp parallel sections num_threads()** para poder seleccionar la cantidad de hilos que queremos que trabajen en nuestros bucles.

CONCLUSIÓN

Utilizando **#pragma omp parallel for**

```
#pragma omp parallel for
{
   for(i=0;i<6;i++)
   {
    for(j=0;j<6;j++)
    // 0/1 generado aleatoriamente
    a [i] [j] = 0 + rand ()% 2;
    }
}

#pragma omp parallel for
{
   for(i=0;i<6;i++)
    for(j=0;j<6;j++)
   {
      if(a[i][j]==1)
    a[i][j]=i*6+j+1;
    }
}
    printf ("Generar puntos de objeto aleatoriamente: \n");
```

En la fase de iniciación no se consigue una mejora, si no que el tiempo de ejecución crece un poco, el tiempo de ejecución pasa a ser de 3.013000 segundos.

En la agrupación utilizando el mismo método también aumenta el tiempo de ejecución el cual es de 3.009000 segundos.

En la actualización de centroides si se consigue una mejora en el tiempo de ejecución el cual es de 3.024000 segundos.

En el tiempo total de ejecución también se consigue una mejora el cual pasa a ser de 3.026000 segundos.

Utilizando **omp parallel sections num_threads** y asignando diferentes hilos para la paralelización no se lograba disminuir el tiempo de ejecución, si no que este subía más.

```
#pragma omp parallel sections num_threads(2)
{
   for(i=0;i<6;i++)
   {
    for(j=0;j<6;j++)
    // 0/1 generado aleatoriamente
    a [i] [j] = 0 + rand ()% 2;
    }
}

#pragma omp parallel sections num_threads(2)
{
   for(i=0;i<6;i++)
    for(j=0;j<6;j++)
   {
      if(a[i][j]==1)
    a[i][j]=i*6+j+1;
    }
}
    printf ("Generar puntos de objeto aleatoriamente: \n");
```

Utilizando **#pragma omp for nowait**

```
#pragma omp parallel for
{
 #pragma omp for nowait
   for(i=0;i<6;i++)
   {

    #pragma omp for nowait
     for(j=0;j<6;j++)
     // 0/1 generado aleatoriamente
     a [i] [j] = 0 + rand ()% 2;
    }
}
```

En la fase de ejecución se consigue disminuir el tiempo de ejecución a comparativa de las pruebas anteriores, siendo el tiempo de ejecución de 3.012000 segundos.

En la agrupación en el tiempo de ejecución aumenta, pasa a ser de 3.019000 segundos.

En la actualización de centroides mantenemos el mismo tiempo, el cual es de 3.024000 segundos.

La ejecución final aumenta a 3.035000 segundos.

El método que nos arrojo los mejores resultados al momento de la ejecución final fuera la primera prueba con **#pragma omp parallel for**.