INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO
Cómputo Paralelo y en la nube

Proyecto Apertura

DBSCAN paralelo para la detección de ruido o outliers con OpenMP

Descripción del código y estrategia de paralelización

Salvador Alejandro Uribe Calva - 188311
Fabio G. Calo Dizy - 191489

En este proyecto se implementó el algoritmo DBSCAN a partir del siguiente pseudocódigo:

```
DBSCAN(D, eps, MinPts)
  C = 0
  for each unvisited point P in dataset D
    mark P as visited
    NeighborPts = regionQuery(P, eps)
    if sizeof(NeighborPts) < MinPts
      mark P as NOISE
    else
      C = next cluster
      expandCluster(P, NeighborPts, C, eps, MinPts)

expandCluster(P, NeighborPts, C, eps, MinPts)
  add P to cluster C
  for each point P' in NeighborPts
    if P' is not visited
      mark P' as visited
      NeighborPts' = regionQuery(P', eps)
      if sizeof(NeighborPts') >= MinPts
        NeighborPts = NeighborPts joined with NeighborPts'
    if P' is not yet member of any cluster
      add P' to cluster C

regionQuery(P, eps)
  return all points within P's eps-neighborhood (including P)
```

En este pseudocódigo, lo que hacemos es visitar todos los puntos si es que no los hemos visitado anteriormente, determinar cuántos vecinos tiene y clasificarlo como ruido o como core de un nuevo clúster a partir de la cantidad de vecinos que tenga. En el caso en el que sí, ahora visitaremos a todos los vecinos que tiene nuestro nuevo clúster y para cada vecino veremos si cumple con las condiciones de ser core. En caso de cumplir la condición, se añade el punto al clúster y se realiza una unión de nuestros vecinos con los vecinos del nuevo miembro del clúster.

El código serial queda de la siguiente manera.

```
void noise_detection(float** points, float epsilon, int min_samples, long long int size) {
   int* cluster = new int[size]; // Tracker de estado de cada punto
   for(long long int i = 0; i < size; i++)
      cluster[i] = 0; // -1 -> Noise // 0 -> Unvisited
                                    // C - Visited, part of cluster C
   int c = 0;
   list<int> vecinos;
   list<int> vecinos2;

   for(long long int i=0; i < size; i++) {
      vecinos.clear();
     if(cluster[i]==0) { //Si no hemos vistado el punto, hacer el proceso
         regionQuery(points, i, epsilon, size, vecinos);
         if(vecinos.size() < min_samples) {
            cluster[i] = -1;
            points[i][2] = 0;
         } else {
            c++;
            cluster[i] = c;
```

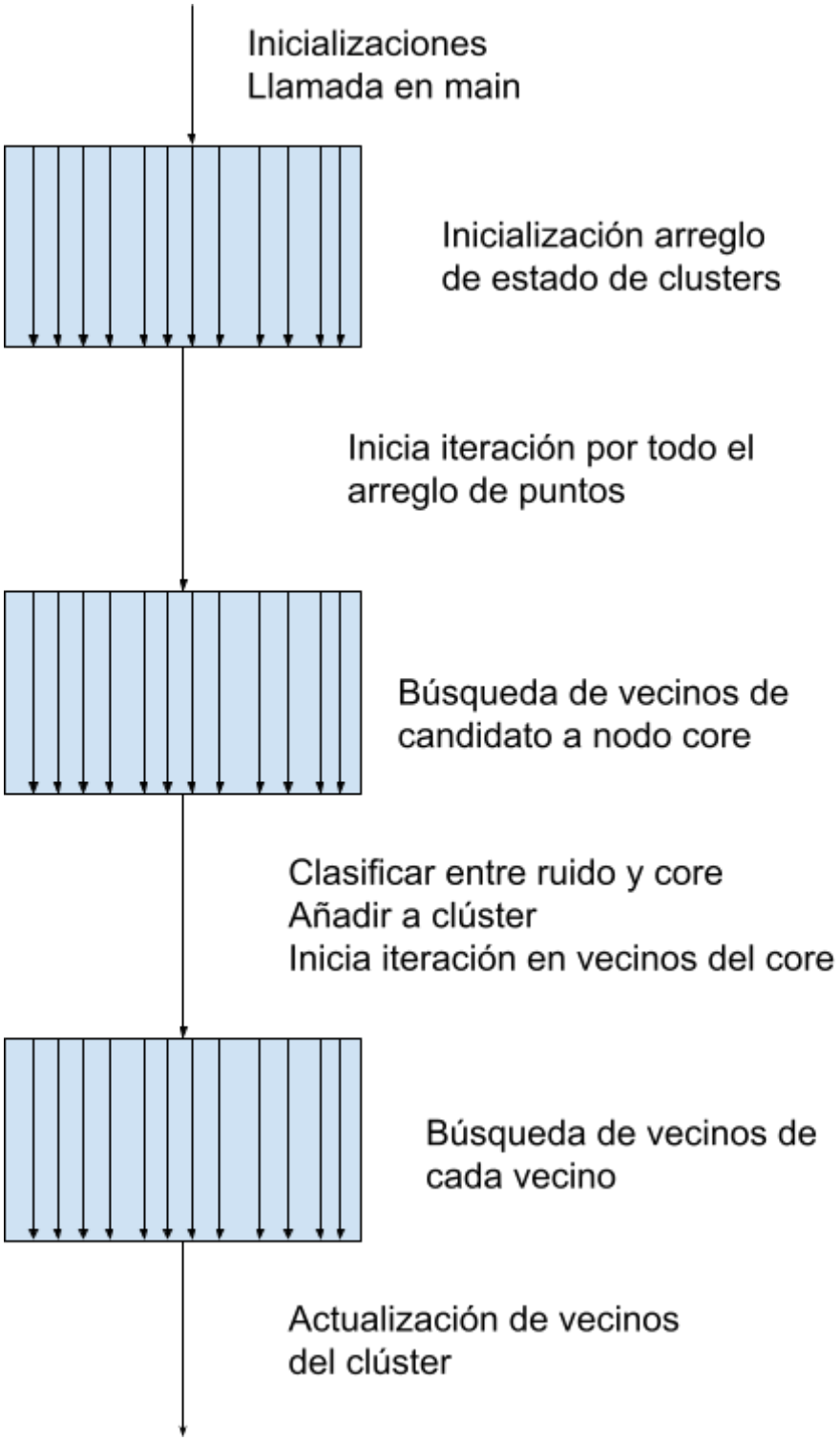
```
            points[i][2] = 1;



            while(!vecinos.empty()) {
                int q = vecinos.front();
                vecinos.pop_front();
                if(cluster[q]==0) {
                    if (cluster[q] <=0 ) {
                        cluster[q] = c;
                        points[q][2] = 1;
                    }
                    regionQuery(points,q,epsilon,size, vecinos2);
                    if(vecinos.size() >= min_samples) {
                      // Union de Stacks
                      vecinos.merge(vecinos2);
                      vecinos2.clear();
                      vecinos.sort();
                      vecinos.unique();
                    }
                }
            }
        }
    }
  }
  cout << c << " Clusters, -> " << "Complete" << "\n";
  delete[] cluster;
}
```

Estrategia de paralelizado

Para paralelizar este código, se pensó en qué partes se podía permitir a los hilos separarse para hacer el trabajo, en qué partes la paralelización era redundante y en qué partes la paralelización traería problemas.



Se intentó que el último bloque de paralelización se iniciara junto con la iteración de los vecinos del core, pero esto terminó dando problemas y creaba más clústers de los que realmente tenía el conjunto de puntos. También hubo problemas al intentar extender la paralelización a la actualización de vecinos debido a la manipulación de listas.

La versión paralela del código quedó de la siguiente manera.

```
void regionQuery(float** points, int p, float epsilon, long long int size, list<int> &vecinos) {
   long long int i;
   float distance;
   float xi = points[p][0];
   float yi = points[p][1];
   # pragma omp for schedule(static)
   for(i = 0; i < size; i++) {
      distance = sqrt(pow(xi- points[i][0],2) + pow(yi- points[i][1],2));
      if (distance < epsilon)
         #pragma omp critical
         {
            vecinos.push_front(i);
         }
   }
}
void noise_detection(float** points, float epsilon, int min_samples, long long int size) {
   int* cluster = new int[size]; // Tracker de estado de cada punto
   int c = 0; // cantidad de clusters
   list<int> vecinos;
   list<int> vecinos2;
   # pragma omp parallel for
   for(long long int j = 0; j < size; j++){
      cluster[j] = 0; // -1 -> Noise // 0 -> Unvisited
   }                              // C - Visited, part of cluster C
   long long int i=0;
   for( i=1; i < size; i++) {
      vecinos.clear();
      if(cluster[i]==0)
      {
         #pragma omp parallel
         {
            regionQuery(points, i, epsilon, size, vecinos);
         }
         if(vecinos.size() < min_samples) {
               cluster[i] = -1;
               points[i][2] = 0;
         }
```

```
        else
        {
            c++;
            cluster[i] = c;
            points[i][2] = 1;

            int q;
            while(!vecinos.empty())
            {
                q = vecinos.front();
                vecinos.pop_front();

                if(cluster[q]==0)
                {
                    if (cluster[q] <=0 )
                    {
                        cluster[q] = c;
                        points[q][2] = 1;
                    }
                    #pragma omp parallel
                    {
                        regionQuery(points,q,epsilon,size, vecinos2);
                    }
                    if(vecinos.size() >= min_samples) {
                        vecinos.merge(vecinos2);
                        vecinos.sort();
                        vecinos.unique();    // Union de Stacks
                        vecinos2.clear();
                    }
                }
            }
        }
    }
std::cout << c << " Clusters, -> " << "Complete" << "\n";
delete[] cluster;
}
```

Adicionalmente, se implementó otra versión de DBSCAN (tanto en serial como en paralelo). La segunda versión tiene dos etapas. Primero, para cada punto se encuentran sus vecinos que están a distancia menor a epsilon. Los puntos se clasifican como *core* si su número de vecinos es mayor o igual que el mínimo de puntos necesario para formar un cluster; en caso contrario, se clasifican como *non-core*.

```
void region_query(float** points, int p, float epsilon, long long int size, vector<int>* neighbors) {
    long long int i;
    float distance;
    float xp = points[p][0];
    float yp = points[p][1];

    // Encuentra todos los vecinos del punto con índice p
    # pragma omp parallel for
    for(i = 0; i < size; i++) {
        //cout << "Number cores " << omp_get_num_threads() << endl;
        distance = sqrt(pow(xp - points[i][0], 2) + pow(yp - points[i][1], 2));

        if(distance < epsilon)
            # pragma omp critical
            {
            neighbors[p].push_back(i);
            }
    }
}
```

```
for(i = 0; i < size; i++) {
    visited[i] = 0;
    // Encuentra todos los vecinos del punto i
    region_query(points, i, epsilon, size, neighbors);

    // Clasifica el punto como core o non-core
    if(neighbors[i].size() >= min_samples)
        core[i] = 1; // 1 -> Core point // 0 -> Non-core point
    else
        core[i] = 0;
}
```

Una vez que se clasifican todos los puntos, se crean los clusters. Para ello, se toma un punto core cualquiera y se añade a un nuevo cluster y a una cola; luego, se visitan los puntos de la cola uno a uno hasta que la cola queda vacía. Cada que se visita un punto, se agregan tanto él como todos sus vecinos al cluster; además, los vecinos que son core entran a la cola. De esta forma, los puntos core expanden el cluster hasta que se encuentran todos los puntos que pertenecen a él.

Una vez se halla un cluster, el algoritmo inspecciona los puntos restantes hasta encontrar un core no visitado. Eventualmente todos los puntos core son visitados, con lo cual el algoritmo acaba y los puntos no visitados quedan clasificados como ruido.

```
for(i = 0; i < size; i++) {
    // Si no se ha visitado el punto i, es parte de un nuevo cluster o es ruido
    if(!visited[i] && core[i] == 1) {
        explore.push_back(i);
        //clusters[num_clusters].push_back(i);

        //# pragma omp critical
        {
        // Explora los vecinos del punto core i y expande el cluster hasta terminarlo
        while(explore.size() > 0) {
            int p = explore.front();
            explore.pop_front();

            for(long long int k = 0; k < neighbors[p].size(); k++) {
                int neighbor = neighbors[p].at(k);

                if(!visited[neighbor]) {
                    visited[neighbor] = 1;
                    //clusters[num_clusters].push_back(neighbor);
                    points[neighbor][2] = 1; // Clustered point

                    // Si es core, se agrega al queue
                    if(core[neighbor] == 1)
                        explore.push_back(neighbor);
                }
            }
        }

        num_clusters++;
        }
    }
}
```

Como puede verse en la primera de las tres imágenes, la sección del código que se paralelizó fue la búsqueda de vecinos. Se intentó paralelizar la sección donde se expanden los clusters, pero se generaba una condición de carrera con *num_clusters* y la cantidad de clusters hallados era demasiado alta. A pesar de ello, se obtuvo un speedup satisfactorio comparado con la versión serial, como se muestra en el documento de evaluación experimental.