Angle-Based Clustering

Parallel Algorithm

Ilaria Malinconico

10 febbraio 2025

Introduzione

L'algoritmo ABC (Angle-Based Clustering) si occupa di clusterizzare i dati all'interno di un dataset tramite una procedura basata sul calcolo di angoli fra i punti. Inizialmente, l'algoritmo identifica nel dataset una piccola parte di punti, che corrispondono ai punti di bordo dei cluster, basandosi sull'angolo esistente fra i punti e i loro vicini. Successivamente, i punti di bordo vengono clusterizzati utilizzando l'algoritmo DBSCAN (Density-Based Spatial Clustering of Applications with Noise). I punti rimanenti, poi, vengono assegnati al cluster del punto di bordo più vicino a loro.

L'algoritmo è composto da tre fasi principali:

- Identificazione dei punti posizionati al bordo dei cluster, tramite il calcolo degli angoli fra i vettori differenza che vanno dai punti ai loro kNN (k Nearest Neighbors);
- Applicazione dell'algoritmo DBSCAN ai soli punti di bordo;
- Assegnazione dei punti interni al cluster del punto di bordo più vicino a loro.

Formule matematiche utilizzate

Angoli fra punti

Gli angoli in uno spazio vettoriale Euclideo reale a dimensione finita $V^R \left(\cong R^d,\ d \in N,\ d \geq 2 \right)$ sono definiti tra ogni coppia di vettori $A,\ B \in V^R$ con

$$cos\theta(A,\ B) = \frac{(A,\ B)_R}{\left|A\right|\left|B\right|}$$

Dove $(A, B)_R = \sum_{k=1}^{d} A_k B_k$ è il prodotto scalare tra i due vettori e $\left|A\right| = \sqrt{(A,\ A)_R}$. Per l'angolo risultante reale $\theta\ (A,\ B)$ vale $0 \leq \theta \leq \pi$.

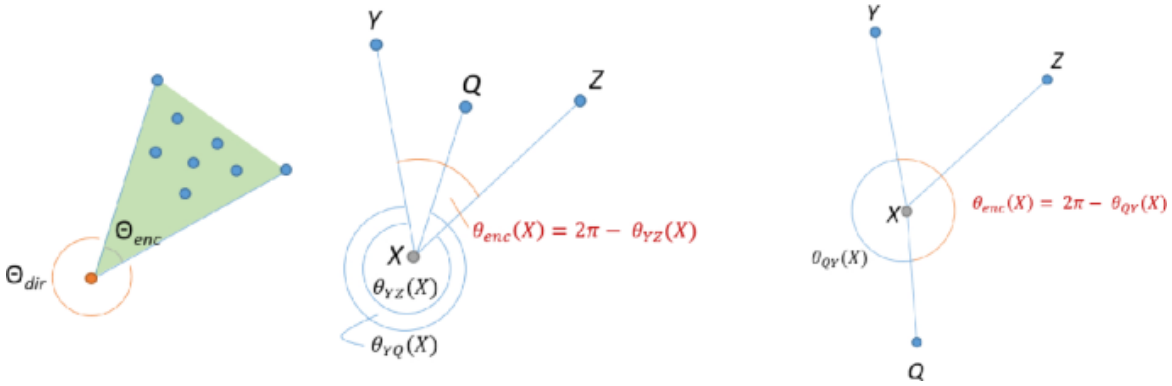
Angoli direzionali e angoli di chiusura

L'angolo di chiusura θ_{enc} di un punto è l'angolo minimo per un punto X tra due vettori differenza, rispetto ai suoi punti vicini, che racchiude tutti gli altri punti vicini (*figura 1*). Un modo per calcolarlo in 2D richiede il calcolo dell'angolo direzionale fra due vettori, $\overrightarrow{XY} = (u_1, u_2), \overrightarrow{XZ} = (v_1, v_2) \in V_2$. L'angolo direzionale in senso antiorario da $\overrightarrow{XY}$ a $\overrightarrow{XZ}$ è $\theta_{YZ}(X) = \arctan 2(u_2, u_1) - \arctan 2(v_2, v_1)$. Se l'angolo risultante è negativo, viene aggiunto 2π in modo da ricevere solo valori positivi fra 0 e 2π.

Per ottenere l'angolo di chiusura di un punto X, si calcola l'angolo direzionale fra i vettori differenza fra tutte le coppie di vicini e si differenziano due casi:

- (*figura 2*) se $\exists Y \in kNN(X) : \forall Z \in kNN(X) : \theta_{YZ} \geq \pi$, allora l'angolo di chiusura si calcola come $2\pi - min(\{\theta_{YZ} | Y, Z \in kNN(X)\})$
- (*figura 3*) altrimenti, l'angolo di chiusura si calcola come $2\pi - max(\{min(\{\theta_{YZ} | Y, Z \in kNN(X)\}) | Y \in kNN(X)\})$

È utile ricordare che i punti posizionati al centro di un cluster tendono ad avere angoli di chiusura più grandi.



Funzione distanza adattata direzione-angolo

Dati due punti di bordo $A, B \in D$, i loro rispettivi vettori direzione a, b e la distanza euclidea fra di essi $d(A,B)_{eucl}$, l'angolo fra i loro vettori direzione $\theta(A,B)$ e un modificatore direzione-angolo σ_{mod}. La distanza fra i due punti di bordo è calcolata come

$$d(A,B)_{mod} = d(A,B)_{eucl} * (1 + (\frac{\sigma_{mod} - 1}{\pi}) * \cos\theta(A,B))$$

Un angolo più ampio fra i vettori direzione a e b risulta in una più grande distanza, dove σ_{mod} controlla il massimo. Un σ_{mod} più alto porta ad una maggiore influenza della similarità direzione-angolo comparata alla distanza euclidea. Quando $\sigma_{mod} = 1$, allora $d(A,B)_{mod} = d(A,B)_{eucl}$. Un valore $\sigma_{mod} < 1$ aumenta la distanza fra punti con angoli diversi.

Implementazione sequenziale

L'algoritmo consiste di tre fasi principali:

1. Dato il dataset di partenza, per ogni punto viene calcolato il grado di bordo. Alla fine del ciclo, i $\beta * N$ punti con il grado di bordo più alto sono i punti di bordo dei cluster;
2. I punti di bordo vengono clusterizzati utilizzando una versione del DBSCAN adattata per l'ABC;
3. I punti interni vengono assegnati allo stesso cluster del punto di bordo più vicino a loro.

Sono stati assegnati valori fissi per β (=0,2) e per k (=12, numero di kNN) estrapolati da [1].

Fase 1

```c
/**
 * Calculate the directional angle between the line segments formed by the center, mean point, and neighbor.
 *
 * @param center     The coordinates of the center point. Struct of two float elements: x, y.
 * @param meanPoint  The coordinates of the mean point.
 * @param neighbor   The coordinates of the neighbor point. Struct of two float elements: x, y.
 * @return The directional angle in degrees.
 */
float getDirectionalAngle(struct double_float center, float *meanPoint, struct double_float neighbor) {
    float uX = 0.00, uY = 0.00, vX = 0.00, vY = 0.00, directionalAngle = 0.00, directionalAngleDegree = 0.00;

    // mean point - center
    uX = meanPoint[0] - center.x;
    uY = meanPoint[1] - center.y;

    // neighbor - center
    vX = neighbor.x - center.x;
    vY = neighbor.y - center.y;

    directionalAngle = atan2(vY, vX) - atan2(uY, uX);
    directionalAngleDegree = directionalAngle * (180 / M_PI);

    if (directionalAngleDegree < 0) {
        directionalAngleDegree += 360;
    }

    return directionalAngleDegree;
}
```

Viene mostrata l'implementazione del calcolo dell'angolo direzionale, utilizzando le differenze di vettori fra il centro, i suoi kNN e i punti medi.

```
/**
 * Calculate the enclosing angle based on the directional angles greater than or equal to 180 degrees.
 *
 * @param directionalAngles The array of directional angles.
 * @return The enclosing angle in degrees.
 */
float getEnclosingAngle(float *directionalAngles) {
    int sizeTmp = findSize(directionalAngles), i;
    float tmpDirectionalAngles[sizeTmp], enclosingAngle = 0.00;
    int counter = 0;
    for (i = 0; i < K; i++) {
        if (directionalAngles[i] >= 180) {
            tmpDirectionalAngles[counter] = directionalAngles[i];
            ++counter;
        }
    }

    float minimumAngle = tmpDirectionalAngles[0];
    for (i = 0; i < sizeTmp; i++) {
        if (minimumAngle > tmpDirectionalAngles[i]) {
            minimumAngle = tmpDirectionalAngles[i];
        }
    }

    enclosingAngle = 360 - minimumAngle;
    return enclosingAngle;
}
```

Viene mostrata l'implementazione del calcolo dell'angolo di chiusura, prendendo come parametri di input l'array di angoli direzionali per il punto in esame.

Viene individuata una soglia di 60° secondo cui, se l'angolo di chiusura è minore di quel valore, allora il punto in esame è un punto di bordo. Infatti, come detto in precedenza, punti posizionati al centro di un cluster hanno angoli di chiusura molto ampi.

```
/**
 * Calculate the border degree based on the minimum directional angle.
 *
 * @param directionalAngles The array of directional angles.
 * @return The border degree in degrees.
 */
float getBorderDegree(float *directionalAngles) {
    float minimumAngle = directionalAngles[0], borderDegree = 0.00;
    for (int i = 0; i < K; i++) {
        if (minimumAngle > directionalAngles[i]) {
            minimumAngle = directionalAngles[i];
        }
    }

    borderDegree = 360 - minimumAngle;
    return borderDegree;
}
```

Viene mostrata l'implementazione del metodo che calcola il grado di bordo, prendendo come parametro di input l'array di angoli direzionali del punto in esame, individuato in precedenza come punto di bordo.

Infine, vengono selezionati i $\beta * N$ punti con il grado di bordo più alto come punti di bordo.

Fase 2

```c
/**
 * Assign cluster labels to the border points based on density connectivity.
 *
 * @param factor               The number of border points.
 * @param epsilon              The distance threshold for neighboring points.
 * @param minNumberPoints      The minimum number of points required to form a cluster.
 * @param borderPointsAndLabels The array to store the cluster labels. Struct of three elements: x (float), y
 (float), label (int).
 */
void getLabelsBorderPoints(int factor, float epsilon, int minNumberPoints, struct point_label
*borderPointsAndLabels) {
    int clusterId = 0, i, j;

    struct triple_float *ptrNeighbors = calloc(factor, sizeof(struct triple_float));
    if (ptrNeighbors == NULL) {
        printErrorAllocation();
    }

    for (i = 0; i < factor; i++) {
        int lenNeighbors = 0;
        if (i != 0) {
            for (j = 0; j < factor; j++) {
                ptrNeighbors[j].x = 0;
                ptrNeighbors[j].y = 0;
                ptrNeighbors[j].z = 0;
            }
        }
        if (borderPointsAndLabels[i].label == 0) {
            lenNeighbors = regionQuery(borderPointsAndLabels, ptrNeighbors, factor, borderPointsAndLabels[i].x,
            borderPointsAndLabels[i].y, epsilon);
            if (lenNeighbors < minNumberPoints) {
                borderPointsAndLabels[i].label = -1;
            } else {
                ++clusterId;
                growCluster(factor, borderPointsAndLabels, i, borderPointsAndLabels[i].x, borderPointsAndLabels[i].
                y, ptrNeighbors, lenNeighbors, clusterId, epsilon, minNumberPoints);
            }
        }
    }

    free(ptrNeighbors);
}
```

Viene mostrata l'implementazione della funzione che clusterizza i punti di bordo e, ad ognuno di essi, assegna una label identificativa. Prende come parametri di input il numero di punti di bordo, la soglia di distanza per i punti vicini, il numero minimo di punti di bordo per formare un cluster e l'array per salvare i risultati ottenuti.

```c
/**
 * Perform a region query to find the neighboring points within a specified distance threshold.
 *
 * @param borderPointsAndLabels The 2D array of border points. Struct of three elements: x
 (float), y (float), label (int).
 * @param neighbors             The 2D array to store the neighboring points. Struct of three
 float elements: x, y, z.
 * @param factor                The number of border points.
 * @param x                     The x component of the point to query.
 * @param y                     The y component of the point to query.
 * @param epsilon               The distance threshold.
 * @return The number of neighboring points found.
 */
int regionQuery(struct point_label *borderPointsAndLabels, struct triple_float *neighbors, int
factor, float x, float y, int epsilon) {
    int counter = 0;
    for (int i = 0; i < factor; i++) {
        float disComputed = directionAngleModifiedDistanceFunction(x, y, borderPointsAndLabels
        [i].x, borderPointsAndLabels[i].y);
        if (disComputed < epsilon) {
            neighbors[counter].x = i;
            neighbors[counter].y = borderPointsAndLabels[i].x;
            neighbors[counter].z = borderPointsAndLabels[i].y;
            ++counter;
        }
    }
    return counter;
}
```

Viene mostrata l'implementazione della funzione che si occupa di calcolare i punti vicini entro una certa soglia di distanza, prendendo come input l'array di punti di bordo, l'array per salvare i punti vicini, il numero di punti di bordo, le coordinate x e y del punto in esame e la soglia di distanza.

```c
/**
 * Calculate the modified distance between two points based on the direction angle between
 their vectors.
 *
 * @param aX The x component of the first point.
 * @param aY The y component of the first point.
 * @param bX The x component of the second point.
 * @param bY The y component of the second point.
 * @return The modified distance between the two points.
 */
float directionAngleModifiedDistanceFunction(float aX, float aY, float bX, float bY) {
    double product = 0.00;
    float angleBetweenVectors = 0.00;
    product = (double) aX * (double) bX + (double) aY * (double) bY;
    angleBetweenVectors = product / (moduleVector(aX, aY) * moduleVector(bX, bY));
    return euclideanDistance(aX, aY, bX, bY) * (1 + ((0.5 - 1) / M_PI) * angleBetweenVectors);
}
```

Viene mostrata l'implementazione della funzione che calcola la distanza adattata per il metodo DBSCAN, che prende in input le coordinate x e y dei due punti fra cui si vuole calcolare la distanza.

```c
/**
 * Grow a cluster by expanding it with neighboring points.
 *
 * @param factor               The number of border points.
 * @param borderPointsAndLabels The array to store the cluster labels. Struct of three elements: x (float), y
(float), label (int).
 * @param index                The index of the initial point.
 * @param x                    The x component of the initial point.
 * @param y                    The y component of the initial point.
 * @param neighbors            The 2D array of neighboring points. Struct of three float elements: x, y, z.
 * @param lenNeighbors         The length of the neighbors array.
 * @param clusterId            The ID of the cluster.
 * @param epsilon              The distance threshold for neighboring points.
 * @param minNumberPoints      The minimum number of points required to form a cluster.
 */
void growCluster(int factor, struct point_label *borderPointsAndLabels, int index, float x, float y, struct
triple_float *neighbors, int lenNeighbors, int clusterId, int epsilon, int minNumberPoints) {
    borderPointsAndLabels[index].label = clusterId;
    int counter = 0, i, j, neighborsIncrement;

    struct triple_float *ptrNextNeighbors = calloc(factor, sizeof(struct triple_float));
    if (ptrNextNeighbors == NULL) {
        printErrorAllocation();
    }

    while (counter < lenNeighbors) {
        int lenNextNeighbors = 0;
        if (counter != 0) {
            for (j = 0; j < factor; j++) {
                ptrNextNeighbors[j].x = 0;
                ptrNextNeighbors[j].y = 0;
                ptrNextNeighbors[j].z = 0;
            }
        }
        int next_index = neighbors[counter].x;
        if (borderPointsAndLabels[next_index].label == -1) {
            borderPointsAndLabels[next_index].label = clusterId;
        } else if (borderPointsAndLabels[next_index].label == 0) {
            borderPointsAndLabels[next_index].label = clusterId;
            lenNextNeighbors = regionQuery(borderPointsAndLabels, ptrNextNeighbors, factor, borderPointsAndLabels
            [next_index].x, borderPointsAndLabels[next_index].y, epsilon);
            if (lenNextNeighbors >= minNumberPoints) {
                neighborsIncrement = 0;
                for (i = 0; i < lenNextNeighbors; i++) {
                    if (checkIfAlreadyNeighbor(neighbors, lenNeighbors, ptrNextNeighbors[i]) == 0) {
                        neighbors[lenNeighbors + neighborsIncrement].x = ptrNextNeighbors[i].x;
                        neighbors[lenNeighbors + neighborsIncrement].y = ptrNextNeighbors[i].y;
                        neighbors[lenNeighbors + neighborsIncrement].z = ptrNextNeighbors[i].z;
                        ++lenNeighbors;
                        ++neighborsIncrement;
                    }
                }
            }
        }
        ++counter;
    }
    free(ptrNextNeighbors);
}
```

Viene mostrata l'implementazione del metodo che si occupa di accrescere un cluster con ulteriori punti vicini. Prende come parametri di

input l'array di punti di bordo, l'indice del punto in esame e le sue coordinate x e y, l'array di punti vicini, il numero di punti vicini, l'id del cluster in esame, la soglia di distanza fra punti vicini e il numero minimo di punti di bordo per formare un cluster.

Fase 3

Infine, vengono individuati i punti interni ai cluster.

```c
/**
 * Assign cluster labels to the non-border points based on distance and labels of border points.
 *
 * @param factor                The number of border points.
 * @param borderPointsAndLabels     The array of cluster labels for border points. Struct of three elements: x
(float), y (float), label (int).
 * @param internalPointsAndLabels   The 2D array of non-border points. Struct of three elements: x (float), y
(float), label (int).
 * @param otherFactor           The number of non-border points.
 */
void getLabelsNonBorderPoints(int factor, struct point_label *borderPointsAndLabels, struct point_label
*internalPointsAndLabels, int otherFactor) {
    float minDistance;
    int labelMin;

    struct double_float *distancesAndLabels = calloc(factor, sizeof(struct double_float));
    if (distancesAndLabels == NULL) {
        printErrorAllocation();
    }

    for (int i = 0; i < otherFactor; i++) {
        minDistance = 0.0;
        labelMin = 0;
        if (i != 0) {
            for (int h = 0; h < factor; h++) {
                distancesAndLabels[h].x = 0;
                distancesAndLabels[h].y = 0;
            }
        }
        for (int j = 0; j < factor; j++) {
            if (borderPointsAndLabels[j].label != -1) {
                distancesAndLabels[j].x = directionAngleModifiedDistanceFunction(internalPointsAndLabels[i].x,
                internalPointsAndLabels[i].y, borderPointsAndLabels[j].x, borderPointsAndLabels[j].y);
                distancesAndLabels[j].y = borderPointsAndLabels[j].label;
            }
        }
        minDistance = findMinimumDistance(distancesAndLabels, factor);
        for (int k = 0; k < factor; k++) {
            if ((minDistance == distancesAndLabels[k].x) && (distancesAndLabels[k].y != -1)) {
                labelMin = distancesAndLabels[k].y;
                break;
            }
        }
        internalPointsAndLabels[i].label = labelMin;
    }

    free(distancesAndLabels);
}
```

Viene mostrata l'implementazione del metodo che clusterizza i punti interni, prendendo come parametri di input il numero di punti di bordo e i loro valori e il numero dei punti interni e i loro valori.

Parallelizzazione

Durante l'analisi teorica di complessità della fase 1 dell'algoritmo sequenziale, non avendo utilizzato strutture di dati avanzate, la complessità nel caso peggiore corrisponde a $O(n^2)$, a causa del doppio ciclo for per il calcolo dei kNN. Infatti, vengono effettuate una serie di operazioni su ogni punto del dataset per individuare i punti di bordo, motivo per cui questa particolare sequenza di istruzioni è stata identificata come porzione di algoritmo sequenziale da parallelizzare, a maggior ragione all'aumentare delle dimensioni del dataset.

In particolare, all'interno del ciclo for:

- Vengono individuati i k-Nearest-Neighbors per ogni punto e il punto di media fra essi;
- Vengono calcolati gli angoli direzionali fra il centro, il punto di media e ogni suo nearest neighbor fra i k individuati;
- Infine, vengono individuati i punti di bordo, con relativo angolo di chiusura per ogni punto, e il grado di bordo.

Una volta che vengono letti tutti i punti del dataset di input, tramite la funzione MPI_Bcast, vengono poi inviati in broadcast dal processo con rango 0 a tutti i processi del comunicatore.

Questi punti, tramite la funzione MPI_Scatterv, vengono poi distribuiti verso ogni processo facente parte del comunicatore, sempre a partire dal processo con rango 0.

Successivamente, viene eseguita la frazione di codice propria della fase 1 discussa in precedenza, in parallelo sul numero di processori specificati in fase di run dell'algoritmo.

Una volta conclusa, viene utilizzata la funzione MPI_Gatherv, che si occupa di raggruppare tutto ciò che i processi nel comunicatore hanno computato.

Infine, la funzione MPI_Reduce si occupa di sommare tutti i localCounter dei singoli processi nel comunicatore, andando ad ottenere il counter complessivo dei punti di bordo.

Le funzioni MPI_Scatterv e MPI_Gatherv sono state preferite ai rispettivi MPI_Scatter e MPI_Gather poiché i primi permettono una maggiore flessibilità per quanto riguarda la distribuzione dei dati fra i processi, nel caso in cui i dati contenuti nel dataset non fossero esattamente divisibili fra il numero di processori identificati durante il run dell'algoritmo.

Prendiamo come esempio pratico il caso in cui il dataset sia formato da 2500 elementi e i processori utilizzati siano 8.

I 2500 elementi dovranno essere divisi fra gli 8 processori, quindi ad ogni processore saranno assegnati

$$chunkSize = 2500/8 = 312$$

elementi, con resto 0,5 da suddividere ulteriormente fra i processori. Di seguito viene mostrato, per ogni processore, il numero di elementi che vengono distribuiti ad ogni processo (counts) e l'offset da applicare sul buffer di invio (displacement)

	0	1	2	3	4	5	6	7
Counts	313	313	313	313	313	312	312	312
Displacement	0	313	626	939	1252	1565	1877	2189

Sono state rilevate delle difficoltà nel parallelizzare la fase 2 - e di conseguenza anche la fase 3 - in quanto, per quanto riguarda l'assegnazione di un cluster ad ogni punto di bordo, il calcolo della funzione di distanza deve essere eseguito come confronto fra un punto di bordo e tutti gli altri. Ciò ha comportato una complicazione nell'individuazione di una strategia ottimale per l'implementazione in parallelo.

Funzioni MPI utilizzate

Documentazione consultata su https://mpi.deino.net/mpi_functions.

int **MPI_Init**(int *argc, char ***argv);

Inizializza l'ambiente di esecuzione di MPI.

- argc: (in) puntatore al numero di argomenti
- argv: (in) puntatore al vettore di argomenti

int **MPI_Comm_rank**(MPI_Comm comm, int *rank);

Determina il rango del processo chiamante nel comunicatore.

- comm: (in) comunicatore
- rank: (out) rango del processo chiamante nel comunicatore

int **MPI_Comm_size**(MPI_Comm comm, int *size);

Determina la dimensione del gruppo di processi associati al comunicatore.

- comm: (in) comunicatore
- size: (out) numero di processi nel gruppo del comunicatore

int **MPI_Barrier**(MPI_Comm comm);

Funge da barriera, blocca tutto finché tutti i processi del comunicatore non hanno raggiunto questa chiamata.

- comm: (in) comunicatore

double **MPI_Wtime**(void);

Restituisce un tempo trascorso sul processore chiamante.

int **MPI_Type_contiguous**(int count, MPI_Datatype old_type, MPI_Datatype *new_type_p);

Crea un tipo di dati contiguo.

- count: (in) conteggio di replica
- old_type: (in) vecchio tipo di dato
- new_type_p: (out) nuovo tipo di dato

int **MPI_Type_commit**(MPI_Datatype *datatype);

Committa il nuovo tipo di dato.

- datatype: (in) nuovo tipo di dato

int **MPI_Bcast**(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

Manda in broadcast un messaggio dal processo padre a tutti gli altri processi del comunicatore.

- buffer: (in/out) indirizzo d'inizio del buffer
- count: (in) numero di entry nel buffer
- datatype: (in) tipo di dato del buffer
- root: (in) rango del processo che manda in broadcast il messaggio
- comm: (in) comunicatore

int **MPI_Scatterv**(void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);

Diffonde diverse parti dello stesso buffer a tutti i processi del comunicatore.

- sendbuf: (in) indirizzo del buffer da inviare
- sendcnts: (in) array di integer che specifica il numero di elementi da mandare ad ogni processo
- displs: (in) array di integer che specifica l'offset da applicare ai dati del buffer di invio
- sendtype: (in) tipo di dato degli elementi del buffer di invio
- recvbuf: (out) indirizzo del buffer di ricezione
- recvcnt: (in) numero di elementi nel buffer di ricezione
- recvtype: (in) tipo di dato degli elementi del buffer di ricezione
- root: (in) rango del processo di invio
- comm: (in) comunicatore

int **MPI_Gatherv**(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);

Raggruppa da tutti i processi in un comunicatore in posizioni specifiche.

- sendbuf: (in) indirizzo d'inizio del buffer di invio

- sendcnt: (in) numero di elementi nel buffer di invio
- sendtype: (in) tipo di dato degli elementi nel buffer di invio
- recvbuf: (out) indirizzo del buffer di ricezione
- recvcnts: (in) array di integer che contiene il numero di elementi ricevuti da ogni processo
- displs: (in) array di integer che specifica l'offset a cui piazzare gli elementi ricevuti nel recvbuf
- recvtype: (in) tipo di dato degli elementi del buffer di ricezione
- root: (in) rango del processo ricevente
- comm: (in) comunicatore

int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Riduce i valori raccolti da tutti i processi in un singolo valore.

- sendbuf: (in) indirizzo del buffer di invio
- recvbuf: (out) indirizzo del buffer di ricezione
- count: (in) numero di elementi nel buffer di invio
- datatype: (in) tipo di dato degli elementi del buffer di invio
- op: (in) operazione di riduzione
- root: (in) rango del processo padre
- comm: (in) comunicatore

int **MPI_Finalize**(void);

Termina l'ambiente di esecuzione di MPI.

Correttezza

La correttezza dell'algoritmo si può verificare tramite:

- Correttezza parziale: l'output che viene prodotto è corretto
- Terminazione: l'algoritmo termina

L'analisi si può suddividere nelle tre fasi individuate in fase di implementazione dell'algoritmo.

Fase 1: identificazione dei punti di bordo. L'obiettivo è verificare che ogni punto di bordo viene identificato correttamente.

Per ogni punto del dataset, il processo:

- Calcola i suoi kNN
- Calcola l'angolo di chiusura fra esso, i suoi kNN e il punto medio
- Identifica o meno il punto come punto di bordo

Il passo base dell'induzione è che n=1, quindi il dataset ha un solo punto e non è possibile che abbia vicini, perciò non può essere punto di bordo, che è il comportamento corretto.

Per il passo induttivo, si suppone che si abbia un dataset di n-1 punti. Aggiungendo un nuovo punto, per quest'ultimo verranno individuati i suoi kNN e l'angolo di chiusura, in modo da stabilire se si tratta o meno di un punto di bordo. Anche in questo caso, il comportamento è corretto.

Fase 2: clustering dei punti di bordo. L'obiettivo è verificare che i punti di bordo vengano clusterizzati.

Per ogni punto di bordo individuato dalla fase 1, il processo:

- Calcola i punti di bordo a lui vicini tramite una funzione di distanza
- Assegna un cluster al punto di bordo

Il passo base dell'induzione è che il punto di bordo sia solo uno, quindi non è possibile che abbia vicini, perciò l'algoritmo avrà un solo cluster, che è il comportamento corretto.

Per il passo induttivo, si suppone che i punti di bordo individuati siano factor-1. Aggiungendo un nuovo punto di bordo, quest'ultimo verrà assegnato ad un cluster. Anche in questo caso, il comportamento è corretto.

Fase 3: clustering dei punti interni. L'obiettivo è verificare che i punti interni vengano clusterizzati.

Per ogni punto interno, il processo:

- Calcola il punto di bordo più vicino
- Assegna al punto interno lo stesso cluster del punto di bordo ad esso più vicino

Il passo base dell'induzione è che il punto interno sia solo uno e gli può essere assegnato un cluster purché ci sia almeno un punto di bordo, che è il comportamento corretto.

Per il passo induttivo, si suppone che i punti interni siano otherFactor-1. Aggiungendo un nuovo punto interno, quest'ultimo verrà associato ad un cluster in base al punto di bordo ad esso più vicino. Anche in questo caso, il comportamento è corretto.

Per quanto riguarda la terminazione, si può affermare che:

- La fase 1 termina dopo un massimo di $O(n^2)$ volte
- La fase 2 itera su factor, che è un valore dipendente da n, quindi finito
- La fase 3 itera su otherFactor, anch'esso dipendente da n e quindi finito. Inoltre gli viene assegnato uno e un solo cluster.

Analisi teorica complessità

L'analisi teorica di complessità si può svolgere suddividendo l'algoritmo nelle tre fasi identificate in precedenza:

1. Computazione dei punti di bordo
2. Clustering dei punti di bordo
3. Clustering dei punti interni

Per quanto riguarda la fase 1, è necessario iterare su ogni punto e, per ognuno, calcolare i suoi kNN, la cui procedura consiste nell'iterare nuovamente su ogni punto. La computazione dei punti di chiusura avviene sempre per ogni punto, ma su un numero ridotto di kNN (k, appunto), quindi la sua componente risulta trascurabile rispetto al calcolo precedente, per cui si può concludere che la fase 1 ha una complessità di $O(n^2)$.

Per quanto riguarda la fase 2, che consiste in un adattamento dell'algoritmo DBSCAN, per ogni punto di bordo (da codice, numero punti di bordo = factor) individuato dalla fase 1 si occupa di trovare i suoi punti di bordo vicini tramite la funzione distanza adattata direzione-angolo e di associare ad ognuno di essi una label che identifica il cluster di pertinenza. Sia la ricerca di vicini, sia l'associazione al cluster di pertinenza sono esecuzioni che iterano su ogni punto di bordo, motivo per cui la complessità in questa seconda fase vale $O(factor^2)$. Considerando che factor viene calcolato in funzione di n, si può concludere che la complessità anche per la fase 2 è $O(n^2)$.

Per quanto riguarda la fase 3, che consiste nell'assegnare ad ogni punto interno (da codice, numero punti interni = otherFactor) un cluster, si può concludere che la complessità di questa fase vale O(factor x otherFactor), in quanto necessita di un'iterazione su ogni punto di bordo per ogni punto interno. Considerando che sia factor sia otherFactor sono calcolati in

funzione di n, si può concludere che la complessità anche per la fase 2 è $O(n^2)$.

È stata sottoposta a parallelizzazione solo la fase 1 dell'algoritmo, in quanto individuata come quella a complessità maggiore nel caso in cui il numero di punti di cui è composto il dataset risulti molto ampio.

È bene sottolineare che, nel caso di utilizzo di strutture di dati più complesse, come un k-d-tree, la complessità dell'algoritmo avrebbe potuto essere migliorata, come dimostrato in [1].

Isoefficienza, scalabilità e cost-optimality

Per isoefficienza si intende il metodo di misura della scalabilità di un sistema parallelo, mentre la scalabilità è la misura dell'abilità di un sistema parallelo di aumentare le performance all'aumentare del numero di processori.

L'efficienza si calcola come il rapporto fra il tempo di esecuzione in sequenziale (T_1) e il prodotto fra il numero di processori (p) e il tempo di esecuzione in parallelo (T_P).

$$E = T_1/pT_p$$

Si analizzano i tempi di esecuzione delle tre fasi dell'algoritmo:

- Fase 1
  - Sequenziale: $O(n^2)$
  - Parallelo: $O(n^2/p)$
- Fase 2
  - Sequenziale: $O(n^2)$ = Parallelo
- Fase 3
  - Sequenziale: $O(n^2)$ = Parallelo

Si può quindi concludere che il tempo totale di esecuzione in parallelo è dato da

$$T_P = O(n^2/p) + O(n^2) + O(n^2) = O(n^2/p) + O(n^2) \approxeq O(n^2)$$

Mentre il tempo totale di esecuzione in sequenziale è dato da

$$T_1 = O(n^2) + O(n^2) + O(n^2) = O(n^2)$$

L'efficienza è dunque data da

$$E = T_1/pT_p = O(n^2)/pO(n^2) = O(1/p)$$

Si può concludere che l'efficienza decresce all'aumentare del numero di processori, risultato dovuto alla parte dell'algoritmo rimasta sequenziale: ciò comporta anche una scalabilità limitata. Infatti, la frazione di algoritmo

parallelizzata permette migliori prestazioni all'aumentare del numero di processori solo fino ad un certo punto, poi il miglioramento rallenta.

Anche se si aumentasse n, le parti di algoritmo rimaste sequenziali sono limitanti, causando un aumento del tempo di esecuzione.

Un sistema parallelo è ottimale in termini di costo se il tempo di esecuzione impiegato da ogni processore, incluso quello di comunicazione, è dello stesso ordine di grandezza del tempo di esecuzione dell'algoritmo sequenziale.

$$pT_p = p(O(n^2/p) + O(logp) + O(n^2)) = O(n^2) + O(plogp) + O(pn^2)$$

Dove O(plogp) è trascurabile in quando riguarda il tempo di comunicazione fra i processori, ma la componente sequenziale all'interno dell'algoritmo parallelo non è trascurabile, rendendolo non ottimale in termini di costo.

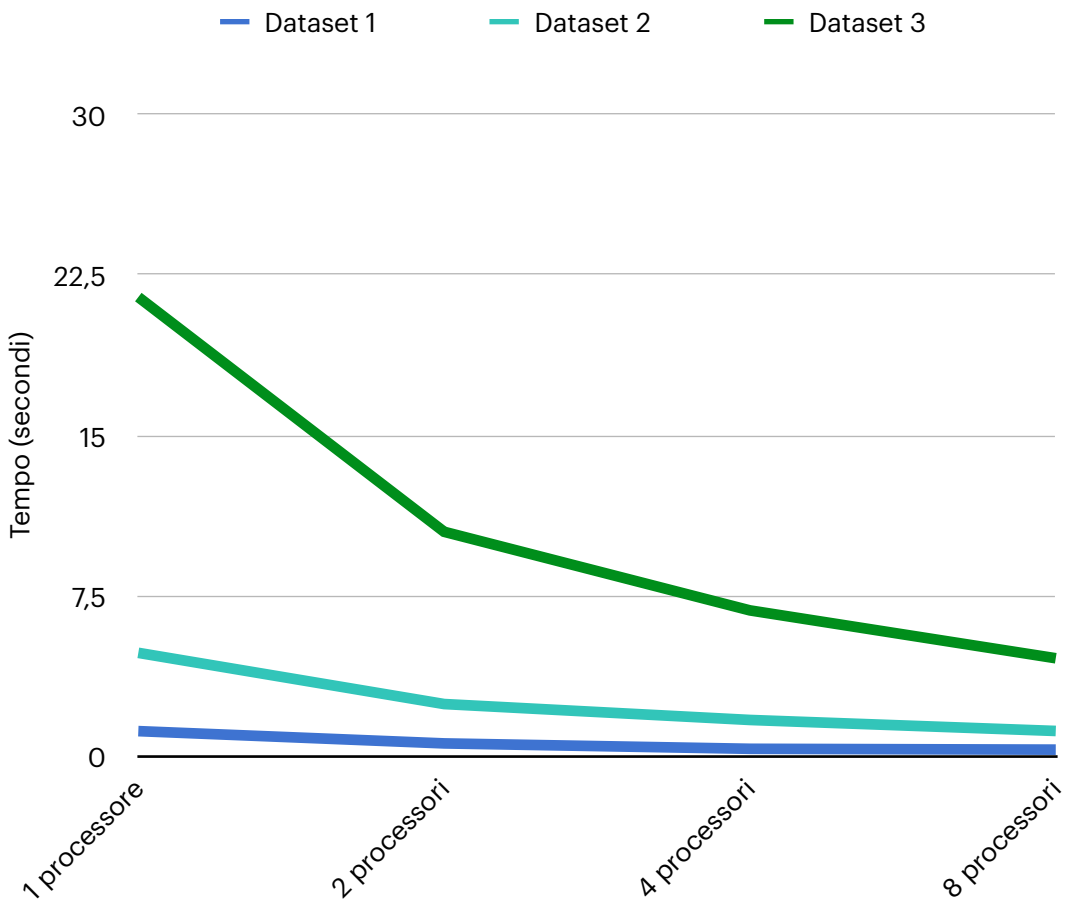
Benchmark

La macchina su cui sono stati eseguiti i test è un MacBook Air con chip M1 del 2020 con CPU 8-core.

I test sono stati effettuati utilizzando tre dataset, composti da un numero di punti differenti, 2500, 5000 e 10000.

L'algoritmo è stato eseguito utilizzando 1, 2, 4 e 8 processori e il tempo di esecuzione della fase 1 parallelizzata è stato riportato nella tabella di seguito.

Dataset	#points	Time (seconds) 1 processor	Time (seconds) 2 processors	Time (seconds) 4 processors	Time (seconds) 8 processors
1	2500	1,152619	0,578111	0,325829	0,286564
2	5000	4,815886	2,420030	1,675185	1,160290
3	10000	21,465422	10,481643	6,802520	4,563172

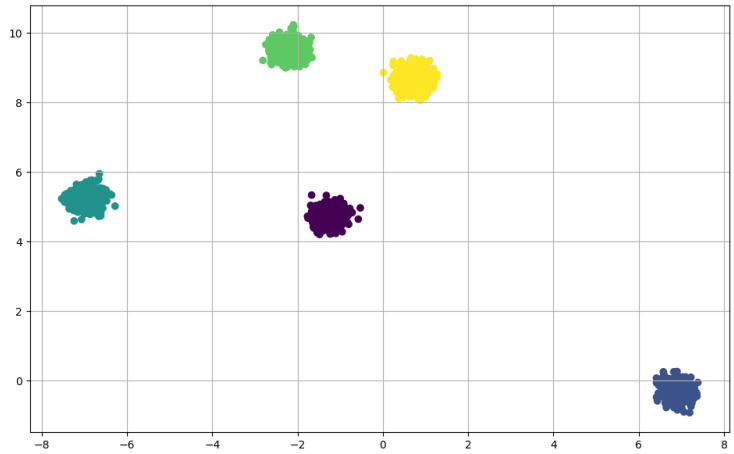


Per quanto riguarda il tempo di esecuzione delle fasi 2 e 3, che vengono eseguite dal solo processo a rango 0 e quindi in sequenziale, al variare del numero di processori rimane sempre orientativamente costante. Infatti:

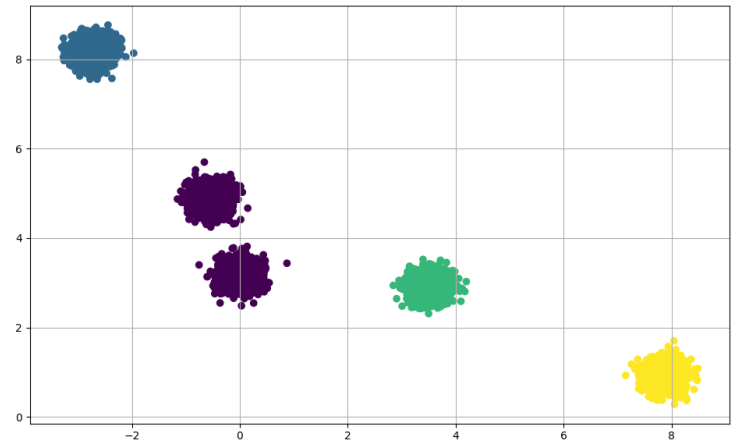
- Per il dataset da 2500 punti, il tempo di esecuzione sequenziale vale circa 0,03 secondi
- Per il dataset da 5000 punti, il tempo di esecuzione sequenziale vale circa 0,1 secondi
- Per il dataset da 10000 punti, il tempo di esecuzione sequenziale vale circa 0,3 secondi

I valori elencati sono trascurabili considerando il numero esiguo di processori su cui sono stati eseguiti i test. È comunque già evidente che, all'aumentare del numero di processori, il tempo di esecuzione sequenziale dell'algoritmo ha sempre maggior peso sul tempo di esecuzione complessivo dell'algoritmo, nonostante l'esecuzione in parallelo della fase 1.

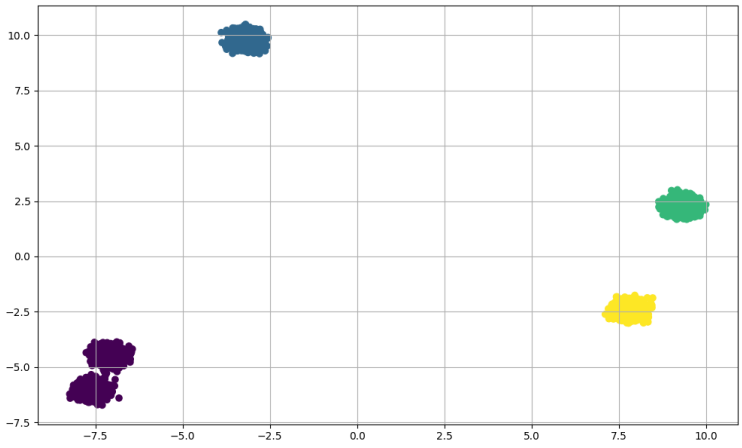
Risultato Clustering Dataset 1



Risultato Clustering Dataset 2



Risultato Clustering Dataset 3



Bibliografia

1. A. Beer, D. Seeholzer, N. S. Schuler, T. Seidl: Angle-Based Clustering