UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN

INFORMATICA(LM-18)

PROGETTO DI HIGH PERFORMANCE COMPUTING:

Federated Parallel Classifier KNN

Relazione a cura di:

Giorgio Aveni
Matricola n° 556126

Dario Miligi
Matricola n° 552033

ANNO ACCADEMICO 2023/2024

INDICE

CAPITOLO 1

INTRODUZIONE

Il progetto in esame costituisce un approfondito esplorare nell'ambito dell'High-Performance Computing (HPC), con l'obiettivo di sviluppare un sistema avanzato di classificazione basato sull'algoritmo k-Nearest Neighbors (KNN). L'aspetto distintivo della nostra implementazione risiede nell'integrazione di un contesto federato nell'esecuzione parallela, una metodologia innovativa che ridefinisce il modo in cui il calcolo distribuito può essere concepito e implementato per ottenere prestazioni ottimali su hardware eterogeneo.

L'aspetto federato di questo sistema si manifesta nella decentralizzazione della gestione del dataset, con ogni nodo del calcolo addebitato dell'indipendente gestione del proprio "chunk" di dati. Questo paradigma federato non solo promuove l'autonomia e l'ottimizzazione delle risorse locali, ma rivoluziona anche il tradizionale modello di distribuzione centralizzata del carico di lavoro, introducendo una nuova era di calcolo distribuito avanzato e altamente efficiente.

L'implementazione sinergica dei paradigmi di programmazione parallela, MPI (Message Passing Interface) e pthreads (POSIX Threads), è stata studiata attentamente per massimizzare l'efficienza computazionale. MPI facilita la comunicazione tra processi, consentendo uno scambio strategico di informazioni, mentre pthreads gestisce la parallelizzazione fine dei thread, ottimizzando le operazioni di calcolo locale. Questa convergenza di tecnologie è stata progettata per ridurre il costo computazionale intrinseco all'algoritmo KNN, consentendo un'elaborazione più rapida e dinamica delle distanze, specialmente in contesti di grandi dimensioni.

Il nostro obiettivo è non solo implementare un sistema di classificazione avanzato ma anche introdurre un paradigma federato che potrebbe rivoluzionare il modo in cui affrontiamo i problemi computazionali su larga scala. Sfide significative emergono durante lo sviluppo, richiedendo soluzioni innovative per affrontare la gestione distribuita dei dati e garantire coerenza e precisione

nell'esecuzione parallela. La relazione esaminerà come l'integrazione di MPI e pthreads offre soluzioni avanzate a tali sfide.

Attraverso test approfonditi e misurazioni delle prestazioni, analizzeremo criticamente l'efficacia delle scelte implementative e valuteremo l'impatto dell'aspetto federato sulla scalabilità complessiva del sistema. L'obiettivo è dimostrare come la decentralizzazione e la gestione distribuita del dataset possano portare a un miglioramento significativo delle prestazioni, soprattutto quando si lavora con grandi dataset.

In conclusione, il nostro progetto non solo si propone di implementare un algoritmo di classificazione avanzato ma introduce un paradigma federato rivoluzionario nell'ambito dell'High-Performance Computing. L'aspetto federato potrebbe delineare nuovi standard di efficienza e scalabilità, non solo per il KNN ma anche per una vasta gamma di applicazioni. La relazione fornirà una visione prospettica sulle potenzialità future di questa proposta, delineando percorsi di miglioramento e raffinamento del sistema alla luce delle sfide uniche dell'HPC.

CAPITOLO 2

ARCHITETTURA E TOOL USATI

Il programma è organizzato in diversi file per migliorare la modularità e la chiarezza del codice:

- **Header files**: I file di intestazione (header file) sono file al cui interno vengono posti prototipi di funzioni, definizione di tipi e di costanti simboliche. Il nostro programma prevede la presenza di due diversi file di intestazione: **header_processes** ed **header_threads**. Entrambi i file possiedono i prototipi di funzioni e la dichiarazione delle costanti utili ai processi (header_processes) ed ai thread (header_thread) con la definizione della struttura dati per gli stessi. Entrambi i file avranno l'estensione **.h** che indica proprio la tipologia del file di header.

- **Functions files**: I file funzioni contengono l'implementazione delle funzioni cui prototipi sono stati dichiarati nei file di header. Il programma prevede anche in questo caso la presenza di due file di funzioni: **processes_functions** e **threads_functions**. Ciascuno dei due file contiene dunque il codice relativo a tutte le funzioni che saranno utilizzate dai processi (processes_functions) e dai threads (threads_functions).

- Il file **mergeSort.h** implementa il codice relativo all'algoritmo di ordinamento mergeSort, che verrà utilizzato dai threads.

- **Knn.c**: il file in questione implementa il codice relativo all'algoritmo knn che verrà eseguito dai thread.

- **Hierarchical_structure.c**: tale file è il main file del programma. In esso è implementata la struttura gerarchica dei processi nodi del cluster MPI.

Per poter sfruttare la modularità offerta dalla suddivisione del codice in diversi file è necessario includere la dipendenza tra i vari file con il comando **#include "nome_file"**. Di seguito vengono riportate le dipendenze:

- I functions files (processes_functions e threads_functions) includono i file di intestazione header_processes ed header_thread, poiché al loro interno sono dichiarati i prototipi delle funzioni che sono implementate nei functions files.

- Il file knn.c include i file header_thread e thread_functions, poiché al loro interno si trovano variabili e funzioni utili ai thread per poter implementare l'algoritmo di machine learning.

- Il main file hierarchical_structure.c include entrambi i file di header, il file processes_functions.c e il file knn.c .

Dunque si può affermare che il programma in questione adotta un modello computazionale di tipo SPMD (single process multiple data) poiché il programma viene eseguito su più nodi MPI, ma con diversi dati per ogni processo nodo del cluster. In tale modo quindi i vari processori del cluster cooperano nell'esecuzione del programma per ottenere i risultati più velocemente, ma eseguendo lo stesso programma. Viene dunque implementato il data parallelism, poiché ogni nodo possiede il proprio dataset locale (aspetto federato). Viene anche implementato il task parallelism, poiché attraverso le direttive di sincronizzazione (controllo sul rank del processo), i nodi si "autoprogrammano" per eseguire istruzioni diverse.

Come anticipato durante il paragrafo relativo all'introduzione, per sviluppare il programma sono stati utilizzati due differenti tool: MPI e Pthread. Nel prossimo paragrafo vedremo nel dettaglio entrambi i tool ed il modo in cui sono stati impiegati per lo sviluppo del programma.

Il progetto è stato totalmente implementato con il linguaggio C utilizzando come ambiente di sviluppo Visual Studio Code.

2.1 MPI

MPI (Message Passing Interface) è un protocollo di messaggistica standardizzato e portabile progettato da un gruppo di ricercatori provenienti dal mondo accademico e industriale, utilizzato per funzionare su una vasta gamma di architetture di calcolo parallelo. Lo standard definisce la sintassi e la semantica di un nucleo di routine di libreria utili a una vasta gamma di utenti che scrivono programmi di messaggistica portabili in C, C++ e Fortran.
Esistono diverse implementazioni di MPI ben testate ed efficienti, molte delle quali sono open source o nel dominio pubblico. Queste hanno favorito lo sviluppo di un'industria del software parallelo e hanno incoraggiato lo sviluppo di applicazioni parallele su larga scala portabili e scalabili.

L'interfaccia MPI mira a fornire funzionalità essenziali di topologia virtuale, sincronizzazione e comunicazione tra un insieme di processi (che sono stati mappati su nodi/server/istanze di computer) in modo indipendente dal linguaggio, con una sintassi specifica del linguaggio (binding) e alcune caratteristiche specifiche del linguaggio. I programmi MPI lavorano sempre con processi, ma i programmatori comunemente si riferiscono a essi come "processori". Tipicamente, per ottenere le massime prestazioni, a ogni CPU (o core in una macchina multi-core) viene assegnato un singolo processo. Questa assegnazione avviene durante l'esecuzione attraverso l'agente che avvia il programma MPI, chiamato normalmente mpirun o mpiexec.

Le funzioni della libreria MPI includono, ma non sono limitate a, operazioni di invio/ricezione di tipo rendezvous punto-a-punto, la scelta tra una topologia logica cartesiana o ad un grafo, lo scambio di dati tra coppie di processi (operazioni di invio/ricezione), la combinazione di risultati parziali di calcoli (operazioni di gather e reduce), la sincronizzazione dei nodi (operazione di barrier) e l'ottenimento di informazioni legate alla rete, come il numero di processi nella sessione di calcolo, l'identità del processore corrente a cui un processo è mappato, i processi vicini accessibili in una topologia logica, e così via. Le operazioni punto-a-punto si presentano in forme sincrone, asincrone, bufferizzate e ready, per consentire sia semantica più forte che più debole per gli aspetti di sincronizzazione di un invio rendezvous. Molte operazioni sono possibili in modalità asincrona nella maggior parte delle implementazioni.

MPI utilizza oggetti chiamati comunicatori e gruppi per definire quali insiemi di processi possono comunicare tra loro. La maggior parte delle routine MPI richiede di specificare un comunicatore come argomento.
MPI_COMM_WORLD viene utilizzato ogni volta che è richiesto un comunicatore: è il comunicatore predefinito che include tutti i processi MPI. All'interno di un comunicatore, ogni processo ha un proprio identificatore unico, un numero intero assegnato dal sistema durante l'inizializzazione del processo. Un rango è a volte chiamato anche "task ID". I ranghi sono contigui e iniziano da zero. Il rango è utilizzato dal programmatore per specificare la fonte e la destinazione dei messaggi, spesso utilizzato condizionalmente dall'applicazione per controllare l'esecuzione del programma (se rank=0 fai questo / se rank=1 fai quello). Le Routine di Gestione dell'Ambiente sono utilizzate per interrogare e impostare l'ambiente di esecuzione MPI, e coprono una varietà di scopi, come l'inizializzazione e la terminazione dell'ambiente MPI, la query dell'identità di un rango, la query della versione della libreria MPI, ecc. Le più comunemente utilizzate includono **MPI_Comm_size**, **MPI_Comm_rank e MPI_Finalize**.

Le Routine di Comunicazione Collettiva più comuni sono **MPI_Barrier**, **MPI_Bcast** e **MPI_Reduce**. Implementazioni ufficiali di MPI includono **MPICH** e **Open MPI**.

Per il progetto sono stati utilizzati il compilatore MPI mpicc ed il comando di esecuzione mpiexec –np nome_programma , dove np indica il numero di processi.

Per lo sviluppo del codice MPI è stato fondamentale dunque per permettere lo scambio di messaggi fra i vari processi, ma soprattutto per stabilire la struttura gerarchica fra i processi del nostro programma. In particolare tale struttura gerarchica viene esposta come segue:

- Il processo con **rank globale 0** viene selezionato come **server centrale**.
- I processi con **rank globale 1,2,3** sono stati individuati come **server intermedi**.
- I restanti processi sono stati individuati come **dispositivi locali**.

Vengono poi formati tre diversi gruppi contenenti ognuno lo stesso numero di dispositivi locali. Ogni gruppo viene amministrato da un server intermedio. Il numero del gruppo rispetta quello del server intermedio:

- Gruppo 0 formato dal server centrale ed i server intermedi.
- Gruppo 1 gestito dal server intermedio 1.
- Gruppo 2 gestito dal server intermedio 2.
- Gruppo 3 gestito dal server intermedio 3.

I gruppi sono stati creati mediante l'utilizzo della funzione primitiva

```
MPI_Comm intermediary_server1;
// Split the communicator into sub-communicators based on intermediary_server_number_associated
MPI_Comm_split(MPI_COMM_WORLD, intermediary_server_number_associated, rank, &intermediary_server1);
```

che permette di dividere il comunicatore in un secondo comunicatore. I parametri di tale funzione sono:

- **MPI_COMM_WORLD**: comunicatore da dividere;
- **Intermediary_server_number_associated**: rappresenta un tag per identificare il comunicatore, tale tag sarà utile per assegnare i processi al gruppo (comunicatore);
- **Rank**: rango del processo;
- **&intermediary_server**: puntatore al nuovo comunicatore;

Tale istruzione viene ripetuta per ogni gruppo. Ogni processo all'interno del gruppo ottiene un nuovo rank. La suddivisione dei processi nei vari gruppi viene fatta grazie al codice della funzione determineGroupNumber(rank) che prende in ingresso il rank del processo e gli assegna 0 se il processo ha un rango compreso tra 0 e 3 (servers) mentre in caso contrario il numero del gruppo viene calcolato in base al rank del processo attuale secondo.

```
if (rank >= 0 && rank <= 3) {
    group_number = 0;
} else {
    group_number = (rank - 4) % 3 + 1;
}
```

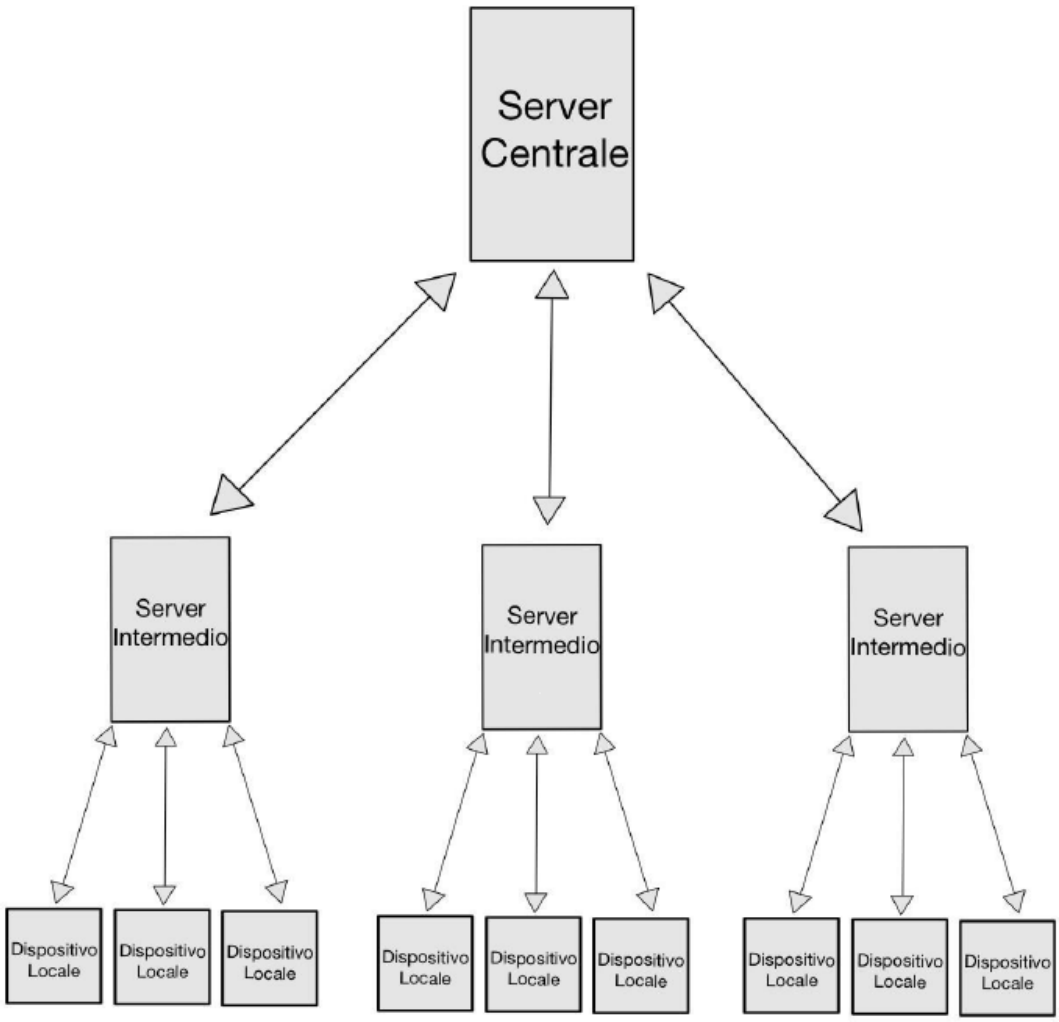
Vengono quindi stampate a schermo tutti i dettagli per ogni processo, quali: rank, ruolo del processo nella struttura gerarchica, numero del gruppo di appartenenza, rank nel gruppo, numero del server associato.

```
Process rank 4.
Local Device.
Group number: 1.
Rank in the group 1: 1.
Intermediary Server associated: 1.
```

```
Process rank 1.
Intermediary Server 1.
Group number: 0.
Rank in the group 1: 0.
Rank in the server group: 1.
```

```
Process rank 0.
Central server.
Rank in the group: 0.
```

Come si evince dalle immagini ogni server intermedio ha rank 0 all'interno del gruppo che gestisce. Segue un'immagine che mostra una panoramica della struttura gerarchica menzionata:



2.2 PTHREAD

La programmazione multithreading è una tecnica avanzata che consente a un programma di eseguire più thread contemporaneamente, migliorando l'efficienza e la capacità di risposta. **pthread** è una libreria standard di interfaccia di programmazione per la gestione dei thread su sistemi operativi POSIX-compatibili. In questa relazione, esploreremo i concetti chiave di pthread e come questo è stato utilizzato nel nostro programma.

Un thread è una singola sequenza di esecuzione all'interno di un processo. Più thread possono condividere lo stesso spazio di indirizzamento e le risorse, ma hanno i propri registri e stack di esecuzione. I processi possono contenere più thread, ciascuno dei quali esegue operazioni indipendenti.

I vantaggi della programmazione mutithreading sono:

Parallelismo: I thread possono eseguire operazioni in parallelo, sfruttando i processori multi-core per migliorare le prestazioni.

Responsività: L'utilizzo di thread separati consente a un'applicazione di rimanere reattiva durante operazioni lunghe o di rete.

Efficienza delle Risorse: La condivisione delle risorse tra thread può ridurre la duplicazione e migliorare l'efficienza.

La funzione **pthread_create** è utilizzata per creare nuovi thread. Richiede la specifica della funzione che il thread eseguirà e altri parametri. Per attendere la terminazione di un thread, si utilizza **pthread_join**.

```
pthread_t threads[NTHREADS];
ThreadData thread_data[NTHREADS];

for (int i = 0; i < NTHREADS; i++) {
    pthread_create(&threads[i], NULL, threadFunction, (void *)&thread_data[i]);
}

for (int i = 0; i < NTHREADS; i++) {
    pthread_join(threads[i], (void **)&thread_data[i]);
    thread_results[i] = thread_data[i].predicted_label;
}
```

La libreria pthread fornisce un'ampia gamma di funzionalità per la programmazione multithreading su sistemi POSIX. La sua corretta implementazione richiede una comprensione approfondita dei concetti di sincronizzazione, mutex, variabili di condizione e gestione degli errori. Utilizzando pthread, è possibile sviluppare applicazioni efficienti, reattive e affidabili, sfruttando al massimo le potenzialità dei sistemi multi-core moderni.

Nel nostro progetto, ogni processo MPI crea al suo interno tre thread, i quali saranno responsabili del calcolo delle distanze tra il punto da classificare e il punto di test, relative all'algoritmo knn. In particolare ogni thread assumerà un determinato numero di righe dal dataset locale del processo, e si occuperà di esercitare l'algoritmo secondo il suo dominio di righe. Ogni thread viene avviato in parallelo per poter avere tempi di risposta più rapidi. I thread avranno tutti la stessa struttura che è stata definita all'interno del file header_thread.h . Vediamo quali sono gli elementi della struttura dei thread:

```c
typedef struct {
    int thread_id;
    int start_row;
    int end_row;
    int num_columns;
    float *data_matrix;
    int *label_matrix;
    float *test_point;
    float *local_distances;
    int predicted_label;
} ThreadData;
```

- **Thread_id**: numero intero che definisce l'identificativo del thread;
- **Start_row**: numero intero che definisce il numero della riga di partenza del dominio di righe del thread;
- **End_row**: numero intero che definisce il numero della riga finale del dominio di righe del thread;
- **Num_columns**: numero di colonne della matrice locale del processo;
- ***data_matrix**: puntatore alla matrice locale di dati float del processo;
- ***label_matrix**: puntatore alla matrice locale di etichette intere del processo;
- ***test_point**: puntatore all'array che rappresenta il punto da classificare;
- ***local_distances**: puntatore alla matrice di distanze calcolate dal thread;

I thread dunque svolgeranno la parte computazionalmente più alta del progetto, ovvero il calcolo delle distanze per l'esecuzione dell'algoritmo knn. In questo modo si ottiene una parallelizzazione fine poiché il lavoro di un singolo processo viene ripartito su più thread, rendendo più veloce l'esecuzione dell'algoritmo.

Un aspetto fondamentale del multithreading è la sincronizzazione dei vari thread. Si parla di sincronizzazione quando è necessario coordinare le azioni di due o più thread per evitare una race condition, ovvero quando due o più thread stanno competendo per accedere a una risorsa condivisa e l'outcome dipende dall'ordine temporale in cui vengono eseguite le operazioni. Per non incombere in tali situazioni è buona pratica utilizzare un mutex, (abbreviazione di "mutual exclusion") ovvero una struttura dati che fornisce un meccanismo di sincronizzazione per garantire l'accesso esclusivo a una risorsa condivisa in un ambiente multiprogrammazione o multithreading.

Nel nostro caso, la sincronizzazione dei thread è stata gestita in maniera diversa, senza l'utilizzo del mutex. Ogni thread possiede un puntatore alle matrici di dati e di etichette del proprio processo padre, pertanto, tutti i thread accederanno a tali risorse per poter assumere le proprie righe su cui effettuare i calcoli (N.B. si è deciso di non creare un'altra matrice all'interno della struttura dei thread e duplicare i dati relativi ad ognuno di questi ultimi poiché, sincronizzando i thread, è possibile che essi operino tutti direttamente sulla matrice locale del processo padre(memory safe)).

Questa situazione potrebbe portare ad un conflitto tra i thread essendo che operano tutti sulle stesse matrici. Il meccanismo di sincronizzazione pensato prevede la dichiarazione di due numeri interi all'interno della struttura dei thread che indicano il numero della riga di inizio e di fine (calcolati dinamicamente) del dominio del thread; in questo modo, anche se più thread accedono contemporaneamente alle matrici del processo, questi non andranno in conflitto durante l'operazione di lettura perché andranno a operare su punti diversi della matrice.

```
for (int i = 0; i < NTHREADS; i++) {
    thread_data[i].thread_id = i;
    thread_data[i].start_row = i * thread_rows + (i < remaining_thread_rows ? i : remaining_thread_rows);
    thread_data[i].end_row = thread_data[i].start_row + thread_rows + (i < remaining_thread_rows ? 1 : 0);
    thread_data[i].num_columns = num_columns;
    thread_data[i].data_matrix = data_matrix;
    thread_data[i].label_matrix = label_matrix;
    thread_data[i].test_point = test_point;
    thread_data[i].local_distances = all_distances;


}
```

Tali valori di inizio e fine vengono passati come parametri di ingresso alla funzione del calcolo della distanza, che viene eseguita da ogni thread. Pertanto ogni thread andrà a calcolare le distanze tra il punto di test e i punti presenti nel intervallo definito da start_row ed end_row.

CAPITOLO 3

FEDERATED LEARNING

Il federated learning è un approccio all'apprendimento automatico decentralizzato e distribuito, progettato per allenare modelli di machine learning senza dover inviare dati sensibili a un server centrale. In un contesto tradizionale di apprendimento automatico, i dati vengono raccolti e inviati a un server centrale per l'allenamento di un modello. Tuttavia, questo solleva preoccupazioni sulla privacy e sulla sicurezza dei dati.

Il federated learning affronta questi problemi permettendo di allenare modelli sui dati locali degli utenti senza la necessità di inviare i dati stessi a un server centrale. Invece, il processo di apprendimento avviene direttamente sui dispositivi degli utenti o su server locali, e solo i pesi del modello vengono inviati e aggregati in modo sicuro. Questo processo riduce il rischio di esposizione di informazioni sensibili e rispetta la privacy degli utenti.

Il processo di Federated Learning si articola in diverse fasi:

1. **Inizializzazione del Modello Centrale**: Un modello iniziale viene definito centralmente e distribuito ai nodi partecipanti.

2. **Addestramento Locale**: I modelli locali vengono addestrati su dati locali senza lasciare il dispositivo. Questo processo comporta il calcolo di gradienti locali basati sui dati locali.

3. **Aggregazione dei Gradienti**: I gradienti calcolati localmente vengono aggregati centralmente senza la necessità di trasferire i dati effettivi. Questo passaggio aggiorna il modello centrale senza mai rivelare i dettagli specifici dei dati locali.

4. **Iterazioni Ripetute**: Il processo di addestramento locale e aggregazione dei gradienti viene iterato fino a raggiungere un modello finale accurato.

Il federated learning è particolarmente utile in scenari in cui la privacy dei dati è critica, come nel caso di dati medici o informazioni personali. Inoltre, può essere applicato in contesti in cui la larghezza di banda è limitata o la connettività è intermittente. Questo approccio consente di beneficiare dell'esperienza di allenamento di un ampio numero di partecipanti senza dover compromettere la sicurezza e la privacy dei dati.

Il nostro progetto implementa una forma di Federated Learning per gestire il dataset distribuito tra i nodi del sistema. In questo contesto, ciascun nodo agisce come un partecipante locale che mantiene il proprio chunk del dataset, evitando la necessità di trasferire i dati centralmente. La comunicazione tra i nodi avviene attraverso la libreria MPI, consentendo l'aggregazione delle informazioni senza la necessità di condividere direttamente i dati di addestramento.

```
void readAssignedRows(FILE *data_file,FILE *label_file,int start_row,int end_row,int num_columns,float *data_matrix,int *label_matrix,char *line_data,char *line_label) {


    // Return the cursor to the beginning of the file
    fseek(data_file, 0, SEEK_SET);
    fseek(label_file, 0, SEEK_SET);

    for (int i = 0; i < end_row; i++) {
        if (i >= start_row) {
            // Read the data
            fgets(line_data, MAX_ROW_LENGTH, data_file);
            char *token_data = strtok(line_data, ",");
            for (int j = 0; j < num_columns; j++) {
                sscanf(token_data, "%f", &data_matrix[(i - start_row) * num_columns + j]);
                token_data = strtok(NULL, ",");
            }

            // Read the labels
            fgets(line_label, MAX_ROW_LENGTH, label_file);
            sscanf(line_label, "%d", &label_matrix[i - start_row]);
        } else {
            fgets(line_data, MAX_ROW_LENGTH, data_file);  // Skip rows not assigned to the current process
            fgets(line_label, MAX_ROW_LENGTH, label_file);
        }
    }
}
```

L'architettura federata del nostro sistema è progettata per garantire una formazione del modello distribuita e sicura. Inizialmente, il modello è distribuito a tutti i nodi partecipanti attraverso il processo MPI. Successivamente, ogni nodo procede con l'addestramento del modello locale, basato sul suo chunk di dati.

L'aggregazione dei modelli locali avviene attraverso operazioni MPI_Gather, che raccolgono i modelli locali a un nodo principale senza mai rivelare dettagli

specifici sui dati. Questo approccio garantisce che il modello centrale venga aggiornato in modo collaborativo senza richiedere il trasferimento diretto dei dati di addestramento.

L'applicazione del Federated Learning nel nostro progetto presenta notevoli vantaggi in termini di sicurezza e privacy dei dati. Poiché i dati rimangono locali a ciascun nodo, non è mai necessario condividere informazioni sensibili o private centralmente. Il processo di addestramento e aggregazione avviene in modo distribuito, mitigando i rischi associati al trasferimento di dati sensibili attraverso la rete.

Inoltre, il coordinamento tra i nodi attraverso MPI assicura che il processo di aggregazione dei modelli avvenga in modo sicuro e controllato. Le operazioni MPI consentono una comunicazione efficiente tra i nodi, contribuendo a garantire la coerenza del modello finale senza compromettere la sicurezza dei dati.

CAPITOLO 4

ALGORITMO KNN

L'algoritmo K-Nearest Neighbors (KNN) svolge un ruolo fondamentale in molteplici applicazioni di machine learning, essendo una tecnica versatile per la classificazione e la regressione. La sua essenza risiede nell'intuizione che le istanze simili sono vicine nello spazio delle feature, e questa vicinanza può essere utilizzata per compiere decisioni di classificazione o stima di valori. Nato come un algoritmo di apprendimento pigro (lazy), KNN evita la fase di addestramento tipica di molti altri algoritmi, imparando invece direttamente dai dati.

Il funzionamento di KNN può essere suddiviso in tre passaggi principali:

1. **Calcolo delle Distanze**: Per classificare un nuovo punto dati, KNN misura la distanza tra questo punto e tutti gli altri punti del dataset. Le metriche di distanza comuni includono la distanza euclidea e la distanza di Manhattan. La scelta della metrica può influenzare significativamente le prestazioni di KNN.

2. **Identificazione dei Vicini**: Una volta calcolate le distanze, KNN identifica i K punti più vicini al punto da classificare. Questi punti costituiscono i "vicini più prossimi" e saranno cruciali per determinare l'etichetta o il valore stimato del punto.

3. **Assegnazione dell'Etichetta o Stimare il Valore**: L'etichetta del punto da classificare è determinata dalla maggioranza delle etichette dei suoi vicini più prossimi nel caso della classificazione. Nel caso della regressione, il valore stimato sarà la media o la mediana dei valori dei vicini.

Algorithm The k-nearest neighbors classification algorithm

Input:

D: a set of training samples $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$
k: the number of nearest neighbors
$d(\mathbf{x}, \mathbf{y})$: a distance metric
$\mathbf{x}$: a test sample

1: **for each** training sample $(\mathbf{x}_i, y_i) \in D$ **do**
2: Compute $d(\mathbf{x}, \mathbf{x}_i)$, the distance between $\mathbf{x}$ and $\mathbf{x}_i$
3: Let $N \subseteq D$ be the set of training samples with the k smallest distances $d(\mathbf{x}, \mathbf{x}_i)$
4: **return** the majority label of the samples in N

Il KNN è noto per il suo costo computazionale elevato, poiché richiede il calcolo delle distanze per ogni punto del dataset. La complessità computazionale è O(N), dove N rappresenta il numero totale di punti nel dataset. Questo rende KNN meno efficiente in scenari in cui il tempo di esecuzione è critico, specialmente su dataset di grandi dimensioni o con un numero elevato di feature.

4.1 ESEMPIO

Dato il seguente dataset:

BRIGHTNESS	SATURATION	CLASS
40	20	Red
50	50	Blue
60	90	Blue
10	25	Red
70	70	Blue
60	10	Red
25	80	Blue

Abbiamo due colonne: Luminosità e Saturazione . Ogni riga nella tabella ha una classe Red o Blue.

Prima di introdurre il nuovo punto da classificare, supponiamo che il valore di K sia 5.

Il punto di test sarà:

BRIGHTNESS	SATURATION	CLASS
20	35	?

Per conoscerne la classe, dobbiamo calcolare la distanza dal punto di test alle a tutti i punti nell'insieme di dati, utilizzando la formula della distanza euclidea:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Dove:

- X_2 : Luminosità del punto da classificare;
- X_1 : Luminosità del punto del dataset;
- Y_2 : Saturazione del punto da classificare;
- Y_1 : Saturazione del punto del dataset;

Dopo aver calcolato le distanze si ottiene la seguente tabella:

BRIGHTNESS	SATURATION	CLASS	DISTANCE
40	20	Red	25
50	50	Blue	33.54
60	90	Blue	68.01
10	25	Red	10
70	70	Blue	61.03
60	10	Red	47.17
25	80	Blue	45

Si riordinano le distanze e si ottiene:

BRIGHTNESS	SATURATION	CLASS	DISTANCE
10	25	Red	10
40	20	Red	25
50	50	Blue	33.54
25	80	Blue	45
60	10	Red	47.17
70	70	Blue	61.03
60	90	Blue	68.01

Poiché abbiamo scelto 5 come valore di K , considereremo solo le prime cinque righe:

LUMINOSITÀ	SATURAZIONE	CLASSE	DISTANZA
10	25	Rosso	10
40	20	Rosso	25
50	50	Blu	33.54
25	80	Blu	45
60	10	Rosso	47.17

Come si evince dalla tabella sopra mostrata, la classe maggioritaria è Red, pertanto al punto di test verrà assegnata tale classe.

CAPITOLO 5

IMPLEMENTAZIONE

Il seguente capitolo mira ad esporre l'implementazione delle versioni parallela e seriale del codice. L'obiettivo principale di questa sezione è fornire una visione dettagliata di come l'algoritmo KNN sia stato implementato utilizzando la parallelizzazione a livello di processo e di thread e di come si presenta una versione seriale del codice.

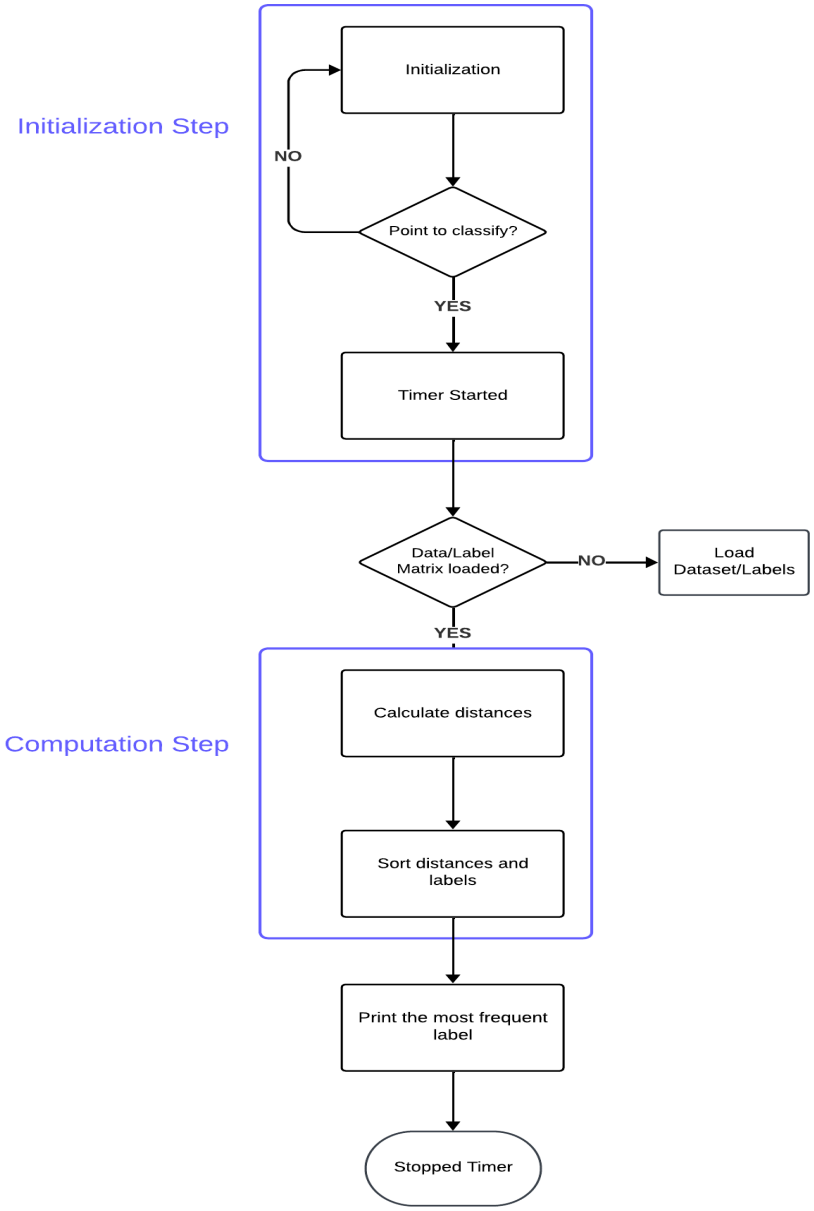
Come già anticipato, il progetto si basa su una struttura federata, in cui ciascun processo mantiene in locale il proprio chunk di dati, consentendo una distribuzione efficiente del carico di lavoro su più nodi e garantendo al contempo una gestione parallela delle operazioni sui thread all'interno di ogni processo. Questa architettura federata contribuisce a migliorare le prestazioni generali del sistema, rendendolo in grado di gestire dataset di grandi dimensioni in modo più efficiente, ed inoltre implementa privacy e sicurezza.

Nel corso di questo capitolo, esploreremo sia l'implementazione seriale che quella parallela del KNN, analizzando come la federazione del dataset tra i processi e l'introduzione dei thread abbia portato a un miglioramento delle prestazioni complessive. Saranno discussi dettagli implementativi specifici, compresi i metodi utilizzati per la gestione delle comunicazioni tra processi, la distribuzione dei dati, la gestione dei thread e l'ottimizzazione del codice per una maggiore efficienza.

Attraverso questa analisi dettagliata, il capitolo sull'implementazione mira a offrire una comprensione completa delle scelte implementative effettuate e delle sfide affrontate durante lo sviluppo del progetto KNN parallelo federato.

5.1 IMPLEMENTAZIONE SERIALE

Innanzitutto, è stata sviluppata un'implementazione seriale dell'algoritmo KNN, la quale offre un punto di riferimento fondamentale per valutare le prestazioni ottenute attraverso la parallelizzazione. In questa versione, il codice è stato strutturato per eseguire in maniera sequenziale l'intero processo di classificazione KNN. È stato mantenuto il linguaggio C per implementare il codice seriale.Il programma legge dati di addestramento da file CSV, richiede all'utente le coordinate di un punto da classificare, calcola le distanze euclidee e restituisce la classe prevista in base alle k vicine. Segue un diagramma di flusso:



Segue un’analisi delle principali funzioni e struttura del codice.

Librerie e Definizioni:

Il codice utilizza librerie standard di C e definizioni per costanti, percorsi dei file, e parametri del classificatore k-NN. Include anche l'header file "mergeSort.h" per la funzione di ordinamento.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mergeSort.h"

#define NFEATURES 4
#define MAX_LINE_LENGTH 1024

#define X_TRAIN_PATH "C:/Users/tecnico/Desktop/UniME/Materie/HPC/MPI/Programs/Parallel-KNN_example/Knn-MPI/X_train.csv"
#define Y_TRAIN_PATH "C:/Users/tecnico/Desktop/UniME/Materie/HPC/MPI/Programs/Parallel-KNN_example/Knn-MPI/y_train.csv"

#define NCLASSES 3
char class[NCLASSES][25] = {"Iris-setosa", "Iris-versicolor", "Iris-virginica"};
#define K 3
```

Inizializzazione dei Dati di Addestramento:

La funzione initializeMatricesFromFile legge dati e etichette da file CSV, contando le righe e inizializzando le matrici. Questo garantisce una corretta preparazione del set di addestramento.

```c
// Function for initializing matrices from CSV files
void initializeMatricesFromFile(char *data_filename, const char *label_filename, float **data_matrix, int **label_matrix, int *num_rows, int num_columns)
```

Stampa delle Matrici:

La funzione printMatrices visualizza le matrici di dati e etichette, offrendo un'opzione di debug per verificare la corretta lettura dei dati.

```c
// Function for printing matrices
void printMatrices(float *data_matrix, int *label_matrix, int num_rows, int num_columns) {
```

Inserimento delle Coordinate del Punto da Classificare:

La funzione inputCoordinates richiede all'utente di inserire le coordinate del punto da classificare, garantendo l'interattività del programma.

```c
// Function to insert the point to be classified
float* inputCoordinates() {
    float* array = (float*)malloc(4 * sizeof(float));

    if (array == NULL) {
        perror("Error allocating memory for the array");
        exit(EXIT_FAILURE);
    }

    printf("\nEnter the coordinates of the point to classify:\n\n");

    printf("Coordinate x: ");
    scanf("%f", &array[0]);

    printf("Coordinate y: ");
    scanf("%f", &array[1]);

    printf("Coordinate z: ");
    scanf("%f", &array[2]);

    printf("Coordinate w: ");
    scanf("%f", &array[3]);

    printf("\n");
    return array;
}
```

Calcolo della Distanza Euclidea:

La funzione calculateEuclideanDistance calcola la distanza euclidea tra due punti nello spazio, fornendo una base essenziale per la classificazione k-NN.

```c
// Function to calculate the Euclidean distance between two points in space
float calculateEuclideanDistance(float point1, float point2) {
    return sqrt(pow(point1 - point2, 2));
}
```

Calcolo delle Somme delle Distanze per Ogni Riga:

La funzione calculateRowSums calcola le somme delle distanze euclidee tra le coordinate del punto da classificare e i punti nel set di addestramento.

```
// Function to add the distances of each coordinate of the point to be classified
float *calculateRowSums(float *data_matrix, int num_rows, int num_columns, float *userPoint) {
```

Ordinamento delle distanze in ordine crescente: nell'intestazione del file è stato incluso il file mergeSort.h che contiene l'algoritmo mergeSort utile per ordinare le distanze e sincronizzare le etichette per ogni punto del dataset.

```
// Call the function to sort in the distances in ascending order and synchronize the labels
mergeSort(rowSums, 0, total_rows-1, label_matrix);
```

Previsione e Stampa della Classe del Punto: La funzione predict determina la classe prevista per il punto in base alle k vicine, calcola le probabilità di appartenenza a ciascuna classe e restituisce la classe più frequente.

```
// Function to predict and print the class for the point
void predict(int *labels) {
    int neighborCount[NCLASSES] = {0};

    for (int i = 0; i < K; i++) {
        neighborCount[labels[i]]++;
    }

    float probability[NCLASSES];
    for (int i = 0; i < NCLASSES; i++) {
        probability[i] = neighborCount[i] * 1.0 / K * 1.0;
    }

    printf("Label      Class      Probability\n");
    for (int i = 0; i < NCLASSES; i++) {
        printf("%d\t%s\t%f\n", i , class[i], probability[i]);
    }

    // Trova la classe più frequente
    int maxClass = 0;
    int maxCount = neighborCount[0];

    for (int i = 0; i < NCLASSES; i++) {
        if (neighborCount[i] > maxCount) {
            maxClass = i;
            maxCount = neighborCount[i];
        }
    }

    // Print the most frequent class
    printf("\nClass for the point: %d - %s\n\n", maxClass, class[maxClass]);
}
```

Funzione Principale (main):

La funzione principale del programma gestisce il flusso di esecuzione. Richiede le coordinate del punto da classificare, legge i dati di addestramento, calcola le distanze, predice la classe e restituisce i risultati e calcola il tempo di esecuzione.

```c
int main() {
    // Dichiarazioni di variabili
    clock_t start_time, end_time;
    double cpu_time_used;

    // Allocazione dei dati
    char *data_filename = X_TRAIN_PATH;
    char *label_filename = Y_TRAIN_PATH;

    int num_rows;
    int num_columns = NFEATURES;
    float *data_matrix;
    int *label_matrix;
    int total_rows = 1000026;

    // Call to the function for inserting the coordinates of the point to be classified
    float *userPoint = inputCoordinates();
    float *distances;

    // Misura l'inizio del tempo di esecuzione
    start_time = clock();

    // Call to the initialization function from CSV file
    initializeMatricesFromFile(data_filename, label_filename, &data_matrix, &label_matrix, &num_rows, num_columns);
    printArray(userPoint);

    // Call to the function for calculating Euclidean distances
    float *rowSums = calculateRowSums(data_matrix, num_rows, num_columns, userPoint);

    // Call the function to sort in the distances in ascending order and synchronize the labels
    mergeSort(rowSums, 0, total_rows - 1, label_matrix);

    // Call to the class prediction function
    predict(label_matrix);

    // Misura la fine del tempo di esecuzione
    end_time = clock();

    // Calcola il tempo di CPU utilizzato
    cpu_time_used = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Stampa il tempo di esecuzione
    printf("Exexution time: %f secondi\n\n", cpu_time_used);

    // Free the allocated memory
    free(userPoint);
    free(data_matrix);
    free(label_matrix);

    return 0;
}
```

L’output finale ottenuto viene mostrato di seguito:

```
Enter the coordinates of the point to classify:

Coordinate x: 4.40
Coordinate y: 3.00
Coordinate z: 1.30
Coordinate w: 0.20

Point to classify: 4.400000 3.000000 1.300000 0.200000

Label      Class      Probability
0       Iris-setosa     0.666667
1       Iris-versicolor 0.333333
2       Iris-virginica  0.000000

Class for the point: 0 - Iris-setosa

Exexution time: 2.178000 seconds
```

5.2 IMPLEMENTAZIONE PARALLELA FEDERATA

Il modello di progettazione dell’algoritmo parallelo utilizzato in questo progetto è il **modello master/slave gerarchico**. Questo è dovuto dal fatto che è presente una struttura gerarchica ed è il server centrale (master di livello superiore) ad inviare ai vari server intermedi (master di secondo livello) il punto da classificare. I dispositivi locali (slave) hanno l’unico ruolo di svolgere i calcoli delle distanze e ordinare le stesse con le etichette di ciascun punto del chunk.

5.2.1 DECOMPOSITION

Per sviluppare correttamente l’algoritmo parallelo, la prima fase consiste nella **decomposizione**. La tecnica di decomposizione consiste essenzialmente nel dividere il problema in dei task che verranno eseguiti in parallelo. Nel nostro progetto, cui obiettivo consiste nella classificazione di un punto mediante l’algoritmo knn, la tecnica utilizzata è la **decomposizione dei dati**, anche se per via del contesto federato non sarà il server centrale a ripartire il dataset fra i vari processi, ma saranno i processi stessi che avranno accesso al file CSV e caricheranno i dati in locale in maniera autonoma, ma ciò non toglie che i dati siano ripartiti fra i vari task. Si è optato per utilizzare una **decomposizione dei dati in input**, poiché l’output corrisponde alla singola etichetta finale per il punto di test, pertanto si può affermare che l’etichetta finale viene ricavata in

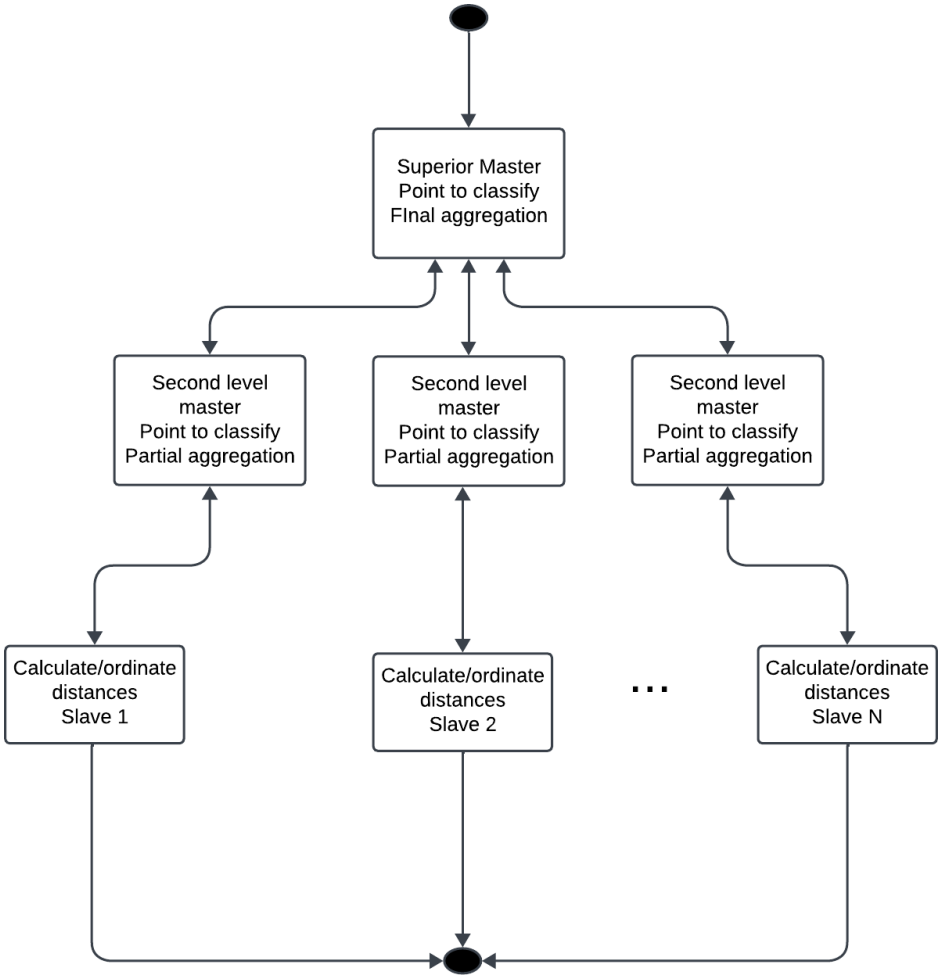
funzione dei dati di input. Questo tipo di decomposizione risulta essere la più naturale, poiché l'output non è conosciuto a-priori. Ogni processo dunque è associato a ciascuna partizione dei dati di input, pertanto ogni task eseguirà il calcolo con la sua parte dei dati. L'elaborazione successiva combinerà tali risultati parziali, difatti, i server intermedi aggregano le etichette di ogni processo del gruppo e selezionano soltanto l'etichetta più frequente, inviandola al server centrale che effettuerà la medesima operazione.

5.2.2 ATTIVITA' DEI TASK

Le strutture utili per la comprensione del comportamento dei task sono il **grafico di dipendenza dei task** e il **grafico di interazione dei task** che mostrano rispettivamente se il lavoro è distribuito su tutti gli slave in qualsiasi momento ese c'è la minima interazione possibile tra i processi.

Nel grafico delle dipendenze dei task, i processi sono rappresentati dai **nodi** del grafico mentre gli archi indicano che il risultato di un task è richiesto per la fase successiva dell'elaborazione. Tale grafico rappresenta le dipendenze di controllo ed è anche un metodo per mostrare la scomposizione. Il grafico dell'interazione dei task rappresenta l'interazione tra le attività coinvolte nel processo: anche in questo caso i nodi rappresentano i processi, ma gli archi rappresentano la loro interazione, ovvero lo scambio di dati tra i due processi. Il suddetto grafico dell'interazione dei task rappresenta le dipendenze dei dati. A volte, è anche possibile che lo stesso grafico possa rappresentare sia il grafico delle dipendenze dei task e il grafico di interazione dei task. Questo è anche il nostro caso, come possiamo vedere in questa sezione.

Come si può ben notare, la comunicazione è bidirezzionale. Questo è dovuto al fatto che i processi sfruttano la struttura gerarchica per scambiare i dati con criterio, difatti la prima comunicazione inizia dal server centrale con l'invio del punto da classificare e continua poi, dopo la fase di calcolo, con l'invio ai vari server intermedi, dell'etichetta predetta da parte degli slave.



Ogni task possiede le seguenti caratteristiche:

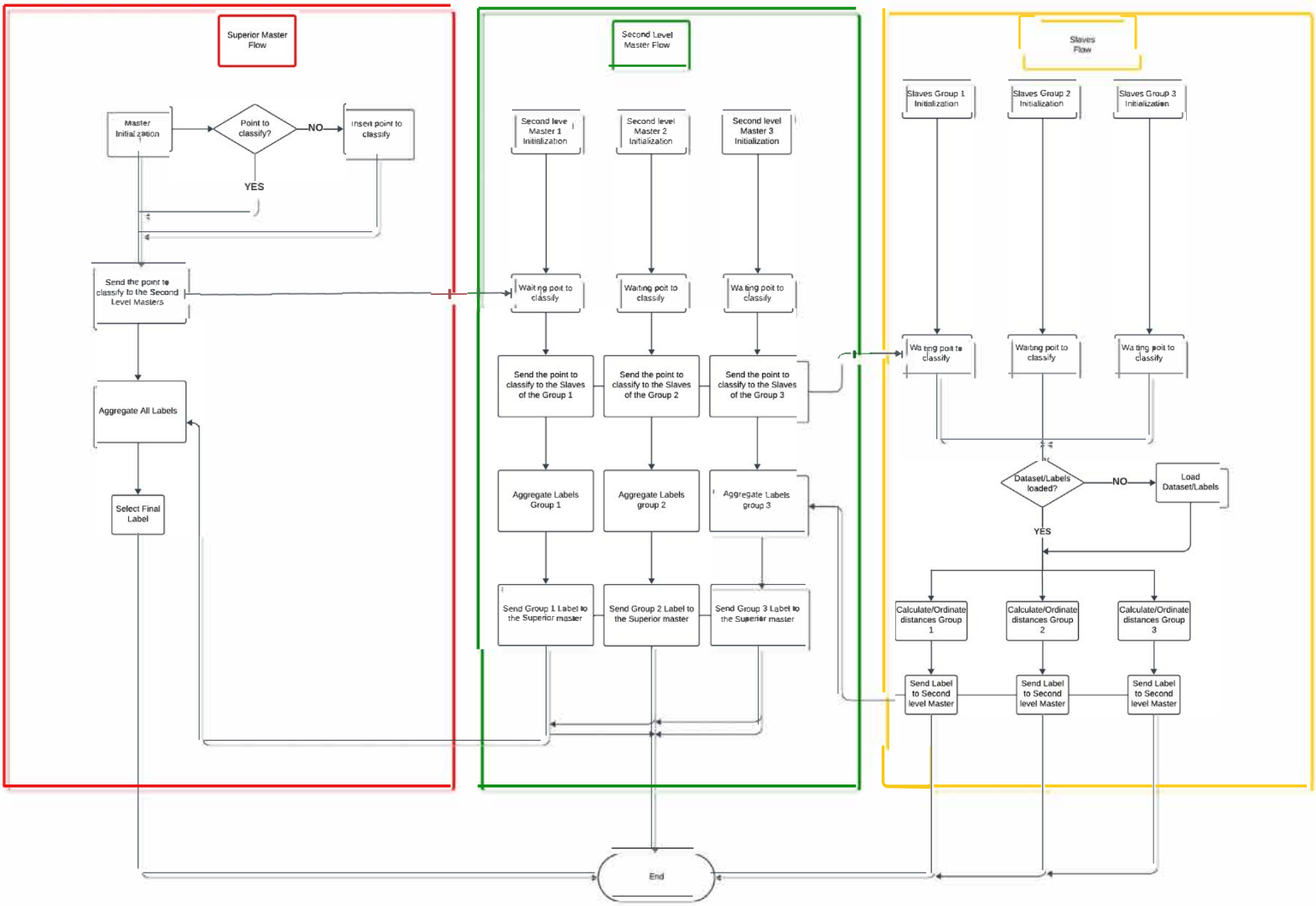
- **Task generation.** Per il progetto abbiamo utilizzato la generazione di attività statiche perché le attività sono definite come parametro di input (come descritto nella sezione 2.1).

- **Task size** che è sostanzialmente uniforme, perché ogni attività occupa una parte del set di dati;

- **Size of data associated to tasks** può essere considerato in un contesto ristretto perché la dimensione del set di dati è inferiore alla quantità di calcoli richiesti per trovare la classe per il punto di test.

5.2.3 MAPPING

La fase di mapping può essere implementata con due differenti tecniche: mapping statico e mapping dinamico. Per il nostro progetto è stato scelto il mapping statico basato sulla partizione dei dati. Lo scopo della fase di mapping è quello di ridurre al minimo il sovraccarico di interazione. I modi per perseguire questo obiettivo sono diversi, come ad esempio quello di massimizzare la località dei dati, che risulta essere pienamente in linea con il contesto federato poiché implica che i dati siano memorizzati localmente nei processi e dunque non è necessaria la comunicazione degli stessi. Un altro aspetto che riduce il sovraccarico di interazione è quello di minimizzare il volume dei dati scambiati fra i vari task, nel nostro caso abbiamo ridotto il volume di tali dati alla sola etichetta predetta che vinene inviata dai vari processi ai rispettivi server.
Il diagramma di flusso seguente mostra il flusso parallelo e l'iterazione tra server centrale, server intermedi e dispositivi locali (superior master, second level masters e slaves). Per prima cosa il server centrale controlla se è stato inserito un punto da classificare, se è presente lo invia ai server intermedi, altrimenti richiede all'utente di inserirlo. Successivamente i server intermedi si occuperanno di inviare il punto da classificare ai dispositivi locali del proprio gruppo di competenza. A questo punto ogni dispositivo locale applica il knn, effettuando il calcolo e l'ordinamento di distanze ed etichette, e inviando al server intermedio del proprio gruppo l'etichetta predetta. I vari server intermedi aggregheranno le etichette ricevute dai dispositivi locali del proprio gruppo, ed invieranno al server centrale l'etichetta più frequente. Il server centrale dunque selezionerà ancora una volta l'etichetta più frequente fra quelle ricevute dai server intermedi, decretando così l'etichetta finale per il punto di test.


Superior Master Flow
Master Initialization
Point to classify?
NO
Insert point to classify
YES
Send the point to classify to the Second Level Masters
Aggregate All Labels
Select Final Label
Second Level Master Flow
Second level Master 1 Initialization
Second level Master 2 Initialization
Second level Master 3 Initialization
Waiting poit to classify
Waiting poit to classify
Waiting poit to classify
Send the point to classify to the Slaves of the Group 1
Send the point to classify to the Slaves of the Group 2
Send the point to classify to the Slaves of the Group 3
Aggregate Labels Group 1
Aggregate Labels group 2
Aggregate Labels group 3
Send Group 1 Label to the Superior master
Send Group 2 Label to the Superior master
Send Group 3 Label to the Superior master
Slaves Flow
Slaves Group 1 Initialization
Slaves Group 2 Initialization
Slaves Group 3 Initialization
Waiting poit to classify
Waiting poit to classify
Waiting poit to classify
Dataset/Labels loaded?
NO
Load Dataset/Labels
YES
Calculate/Ordinate distances Group 1
Calculate/Ordinate distances Group 2
Calculate/Ordinate distances Group 3
Send Label to Second level Master
Send Label to Second level Master
Send Label to Second level Master
End

5.2.4 IMPLEMENTAZIONE

L'implementazione parallela del sistema di classificazione KNN federato si basa sulla convergenza di due paradigmi di programmazione parallela: MPI (Message Passing Interface) e Pthreads (POSIX Threads). Questa combinazione consente di sfruttare appieno le capacità di calcolo distribuito e la parallelizzazione fine dei thread.

Come è stato detto nel capitolo 2.1, MPI è utilizzato per facilitare la comunicazione tra i processi, permettendo lo scambio efficiente di informazioni necessarie durante l'esecuzione parallela. Nel nostro contesto, MPI è impiegato per dividere il comunicatore globale in sotto-comunicatori basati sul gruppo associato, creando così una struttura gerarchica fra i processi. Questa suddivisione consente ai processi di comunicare in modo più efficiente all'interno dei loro sottogruppi, riducendo al minimo il traffico di rete e migliorando le prestazioni complessive.

Pthreads è utilizzato per la parallelizzazione fine dei thread, permettendo di sfruttare le capacità di calcolo locali di ciascun nodo. Riprendendo anche in questo caso i concetti del capitolo 2.2, ogni processo MPI crea un numero specificato di thread Pthreads per gestire la computazione parallela dei dati locali.

L'integrazione di MPI e Pthreads richiede un'attenta gestione delle risorse condivise e una sincronizzazione accurata tra i thread. In particolare, le comunicazioni MPI avvengono tra i processi, mentre Pthreads gestisce la parallelizzazione all'interno di ciascun processo.

L'aspetto federato del sistema si traduce nella decentralizzazione della gestione del dataset. Ogni nodo di calcolo è autonomo nella gestione del proprio "chunk" di dati, consentendo una maggiore efficienza e riducendo la dipendenza da una gestione centralizzata e soprattutto implementando la privacy e la sicurezza non condividendo i dati di addestramento

Il flusso di lavoro del progetto è implementato nel file hierarchical_structure.c e comincia con l’inizializzazione dell’ambiente MPI utilizzando le primitive menzionate nel capitolo 2.1.

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Il primo passo sarà quello di assegnare al processo corrente il numero del proprio gruppo (vedi capitolo 2.1). Successivamente si procederà con la prima suddivisione del comunicatore globale nel comunicatore relativo al gruppo dei server. È stato scelto di creare un gruppo unico per tutti i server per facilitare la comunicazione degli stessi. Il numero di tale gruppo sarà 0

```
int group_number;
int intermediary_server_number_associated;

// Determine the group number for the current process
group_number = determineGroupNumber(rank);

MPI_Comm server_group;
// Split the communicator into subgroups based on group_number
MPI_Comm_split(MPI_COMM_WORLD, group_number, rank, &server_group);

int group_server_rank;
int group_server_size;

// Get information about the subgroup
getGroupInfo(server_group, &group_server_rank, &group_server_size);
```

Si procede poi con l'inizializzazione delle coordinate per il punto da classificare. Questa operazione viene effettuata soltanto dal server centrale, che si occuperà di inviare in broadcast (utilizzando la primitiva MPI_Bcast) il puntoinserito dall'utente ai vari server intermedi tramite il comunicatore del gruppo 0.

```
// Initialization of global message for rank 0
float global_message[4];
if (rank == 0) {
    initializeGlobalMessage(global_message, rank, group_server_rank);
}

// Broadcast the global message to all processes in the subgroup
if (rank >= 0 && rank <= 3) {
    MPI_Bcast(&global_message, 4, MPI_FLOAT, 0, server_group);
}
```

Segue poi la suddivisione dei processi nei restanti gruppi con conseguente invio del punto da classificare in broadcast da parte del server intermedio per il proprio gruppo. L'operazione è stata eseguita con la medesima strategia per tutti i gruppi.

A questo punto in cui ogni processo ha il proprio ruolo all'interno della struttura gerarchica ed ha conoscenza sul punto da classificare, è necessario assumere per ogni processo la propria porzione di dati di addestramento e di etichette provenienti dai file CSV. Il flusso di lavoro per questa determinata operazione è implementato nel file knn.c, il quale include il file funzioni thread_function.c all'interno del quale si troverà l'intera implementazione della logica utilizzata per permettere ad ogni processo di assumere in locale la propria porzione di file CSV.

```
int knn(int rank, int size, char *data_filename, char *label_filename, float *test_point) {
    if (rank > 3) {
        char line_data[MAX_ROW_LENGTH];
        char line_label[MAX_ROW_LENGTH];

        FILE *data_file = fopen(data_filename, "r");
        FILE *label_file = fopen(label_filename, "r");

        openAndCheckCSVFiles(data_filename, label_filename, &data_file, &label_file);

        int total_rows_data = countTotalRows(data_file, line_data);
        int total_rows_label = countTotalRows(label_file, line_label);

        checkEqualNumberRows(total_rows_data, total_rows_label);

        int start_row, end_row;
        calculateRowRange(rank, size, total_rows_data, &start_row, &end_row);

        int num_rows, num_columns;

        float *data_matrix, *all_distances;

        int *label_matrix;

        allocateMatrix(start_row, end_row, &num_columns, &data_matrix, &label_matrix, &all_distances, &num_rows);

        readAssignedRows(data_file, label_file, start_row, end_row, num_columns, data_matrix, label_matrix, line_data, line_label);
```

Come si evince dall'immagine, è stata sviluppata una funzione knn, la quale ingloba l'intero algoritmo, dal caricamento locale per ogni processo delle matrici di dati e di etichette provenienti dai file CSV, fino alla predizione della classe per il punto di test. L'aspetto federato viene implementato proprio in questa situazione, poiché sarà ogni processo che localmente assume il proprio dataset di addestramento, che verrà sempre mantenuto in locale senza mai condividerlo con nessun altro processo.

Una volta che il processo avrà assunto localmente la propria matrice di dati e di etichette proveniente dai file CSV, esso creerà un determinato numero di thread (nel nostro caso 3), con una struttura ben definita (capitolo 2.2). Ogni thread verrà inizializzato tramite un'apposita funzione, popolando così la propria struttura per i calcoli che dovrà svolgere.

```
pthread_t threads[NTHREADS];
ThreadData thread_data[NTHREADS];

initializeThreads(threads, thread_data, num_rows, num_columns, data_matrix, label_matrix, test_point, all_distances);
```

A questo punto, come mostrato nel capitolo relativo a Pthread, vengono creati i thread, i quali eseguiranno tutti una funzione specifica, chiamata thread_function. Questa funzione richiama al suo interno tutte le funzioni utili ai thread per implementare il calcolo delle distanze e l'ordinamento delle stesse.

```
void *threadFunction(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    calcDistance(data);
    int num_rows = data->end_row - data->start_row;
    mergeSort(data->local_distances, 0, data->end_row - data->start_row - 1, data->label_matrix);
    pthread_exit((void *)data);
}
```

Come anticipato in precedenza, i thread nella propria struttura mantengono un puntatore alla matrice di dati ed etichette del proprio processo padre. Essi mantengono nella struttura anche due numeri interi che, come anticipato nel capitolo 2.2 sono utili per sincronizzare i thread ed evitare race condition, poiché tali numeri rappresentano l'inizio e la fine dell'intervallo di punti di addestramento sul quale il thread deve applicare il calcolo delle distanze. Tali estremi saranno utili nella funzione che calcola le distanze euclidee (secondo la formula menzionata nel capitolo 4.1) tra i punti appartenenti al dominio del thread e il punto da classificare.

```
void calcDistance(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    for (int i = data->start_row; i < data->end_row; i++) {
        float sum_distance = 0.0;
        for (int j = 0; j < data->num_columns; j++) {
            float test_point_value = data->test_point[j];
            float point_distance = sqrt(pow(data->data_matrix[i * data->num_columns + j] - data->test_point[j], 2));
            sum_distance += point_distance;
        }
        data->local_distances[i - data->start_row] = sum_distance;
    }
}
```

Una volta calcolate le distanze, queste verranno ordinate, insieme alle relative etichette, grazie all'algoritmo mergeSort, implementato nel file mergeSort.h.

Successivamente all'ordinamento, si procederà con la copia delle distanze ordinate di ciascun thread dentro un array che incorporerà tutte le distanze calcolate da tutti i thread del processo. Tale array sarà disponibile al processo, che lo ordinerà utilizzando sempre la funzione mergeSort e individuerà

l’etichetta più frequente tra le k etichette calcolate dai thread. La funzione knn restituirà dunque l’etichetta predetta dal processo.

```
// Copies all distances calculated by threads into the distance array for the process
int index = 0;
for (int i = 0; i < NTHREADS; i++) {
    pthread_create(&threads[i], NULL, threadFunction, (void *)&thread_data[i]);

    for (int j = 0; j < 5; j++)
    {
        pthread_join(threads[i], (void **)&thread_data[i]);
        thread_distances[index] = thread_data[i].local_distances[j];
        thread_labels[index] = thread_data[i].label_matrix[j];
        index++;
    }
}

mergeSort(thread_distances, 0, (NTHREADS*5)-1, thread_labels);

// Calculate the most frequent label
int *label_counts = calloc(num_rows, sizeof(int));
int max_label = -1, max_count = 0;

for (int i = 0; i < NTHREADS; i++) {
    for (int j = 0; j < NTHREADS*5; j++)
    {
        int current_label = thread_labels[i];
        label_counts[current_label]++;
        if (label_counts[current_label] > max_count) {
            max_count = label_counts[current_label];
            max_label = current_label;
        }
    }
}

free(thread_distances);

cleanupAndClose(data_file, label_file, data_matrix, label_matrix);

return max_label;
```

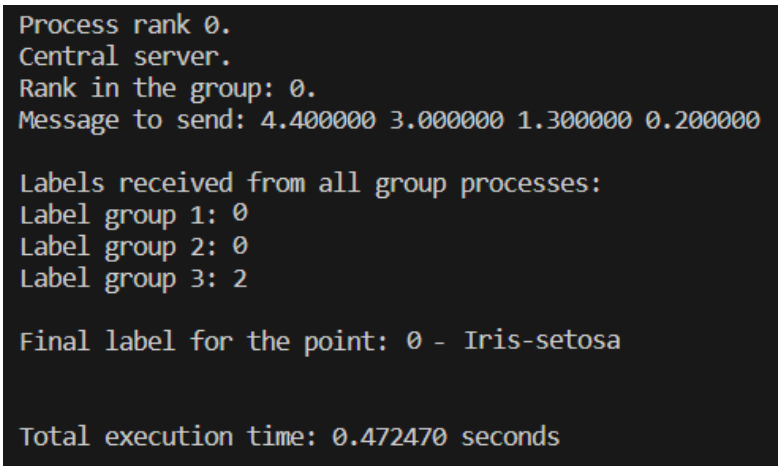
Ritornando al file hierarchical_structure.c i server intermedi procederanno con l’operazione dei raccolta delle etichette predette dai processi dei propri gruppi.

```
int label_intermediary_server1 = 0;
int label_intermediary_server2 = 0;
int label_intermediary_server3 = 0;

// Calculate and print average values for each intermediary server
if (intermediary_server_number_associated == 1) {
    label_intermediary_server1 = calculateAndPrintAverage(rank, gathered_array1, label_intermediary_server1, group1_size);
}
if (intermediary_server_number_associated == 2) {
    label_intermediary_server2 = calculateAndPrintAverage(rank, gathered_array2, label_intermediary_server2, group2_size);
}
if (intermediary_server_number_associated == 3) {
    label_intermediary_server3 = calculateAndPrintAverage(rank, gathered_array3, label_intermediary_server3, group3_size);
}
```

Come accaduto in ogni processo, anche in questo caso i server intermedi si occuperanno di evidenziare la classe più frequente fra quelle del proprio gruppo.

A questo punto non rimane altro che raccogliere tutte le etichette di ogni gruppo calcolate dai server intermedi nel server centrale e ancora una volta selezionare l'etichetta più frequente, che sarà l'etichetta definitiva per il punto di test. Segue l'output finale del programma.



CAPITOLO 5

CASE STUDY

Il caso di studi su cui ci siamo concentrati riguarda la classificazione di fiori iris. Sono state istanziate tre classi con tre relative etichette, che rappresentano diverse tipologie di Iris:

```
// Array of all classes/ label names
char classLabels[NCLASSES][25] = {"Iris-setosa", "Iris-versicolor", "Iris-virginica"};
```

Pertanto ogni classe avrà le seguenti etichette:

Iris-setosa = 0;

Iris-versicolor = 1;

Iris-virginica = 2;

Sono stati creati dieci diversi dataset con dimesioni diverse contenenti un numero crescente di punti, in modo tale da poter effettuare le misurazioni sul tempo di esecuzione per ogni dataset. Naturalmente ad ogni dataset di punti corrisponde un dataset di relative etichette per ogni dato. Ciascun punto del dataset quindi contiene svariate istanze delle classi prima menzionate. I dataset sono stati creati da noi in maniera randomica, tramite due semplici script in Python, come mostrato nella figura successiva.

Ciascun punto del dataset possiede quattro arrtibuti:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

Per ogni valore del di ciascun punto del dataset sono stati stabiliti un intervallo nel quale generare i dati, in modo tale da rimanere il più fedeli possibile ad un ipotetica misurazione reale. È stato utilizzato lo stesso script per creare tutti i

dataset di dimensione diversa, viene infatti cambiato man mano il valore che indica la quantità di dati da generare.

Dataset di dati

```python
import csv
import random

# Set seed for reproducibility
random.seed(42)

#Define the maximum and minimum value ranges for each column
min_values = [4.40, 2.0, 1.0, 0.1]
max_values = [7.70, 4.20, 6.90, 2.50]

# Generates an nx4 matrix of random float numbers within the specified ranges
data = [[random.uniform(min_val, max_val) for min_val, max_val in zip(min_values, max_values)] for _ in range(1000026)]

# Write data to a CSV file
with open('dataset1000000.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    for row in data:
        writer.writerow(row)
```

Per le etichette è stato eseguito uno script pressoché simile, con la differenza che non sono stati decretati intervalli di massimo e minimo, poiché ogni etichetta può assumere soltanto i valori 0,1,2. Anche in questo caso viene utilizzato lo stesso script per creare tutti i dataset di dimensione diversa, aumentando quando è necessario il numero di dati da creare.

Dataset di etichette

```python
import csv
import random

# Generates a list of n integer random numbers between 0 and 2
labels = [random.randint(0, 2) for _ in range(1000026)]

# Specifica il nome del file CSV
file_csv = 'label1000000.csv'

# Scrivi la lista di numeri nel file CSV
with open(file_csv, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(map(lambda x: [x], labels))

print(f'Create the file CSV: {file_csv}')
```

I dati nei file csv si presentano nella seguente maniera:

Dataset di dati

	A	B	C	D	E	F	G	H
1	6.510108434911016,2.055023661489867,2.6226729783778038,0.6357057715571746							
2	6.830355006741241,3.488738872330405,6.263859449458589,0.3086531983105988							
3	5.792342004961393,2.065553882763755,2.28996405134126,1.3128526914480698							
4	4.48756869995675,2.437442831510627,4.834318182899187,1.40785955344772							
5	5.1274540527343,3.2963845045269995,5.775639694399177,0.11559702322734644							
6	7.059203531048267,3.535906668974099,3.007478047456152,0.47315079954827577							
7	7.558803138282379,2.7405079992477788,1.5472004759428728,0.3321193044003136							
8	7.196731408946618,3.3281972690071604,5.762056812318844,1.851356288065163							
9	6.169552701800512,4.140854680754615,3.233352825529286,1.4248975150557448							
10	7.137035392034884,3.3607434552013413,6.084070711833586,1.485645148616229							

Dataset di etichette

	A
1	0
2	0
3	0
4	2
5	0
6	0
7	0
8	0
9	1
10	0

CAPITOLO 6

RISULTATI SPERIMENTALI

Il KNN sequenziale e quello parallelo federato sono stati implementati utilizzando il linguaggio C. Il codice viene eseguito su HP ProDesk 400 G1 MT, Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz 3.20 GHz, RAM 8GB DDR3, 64-bit Windows 10 Pro.

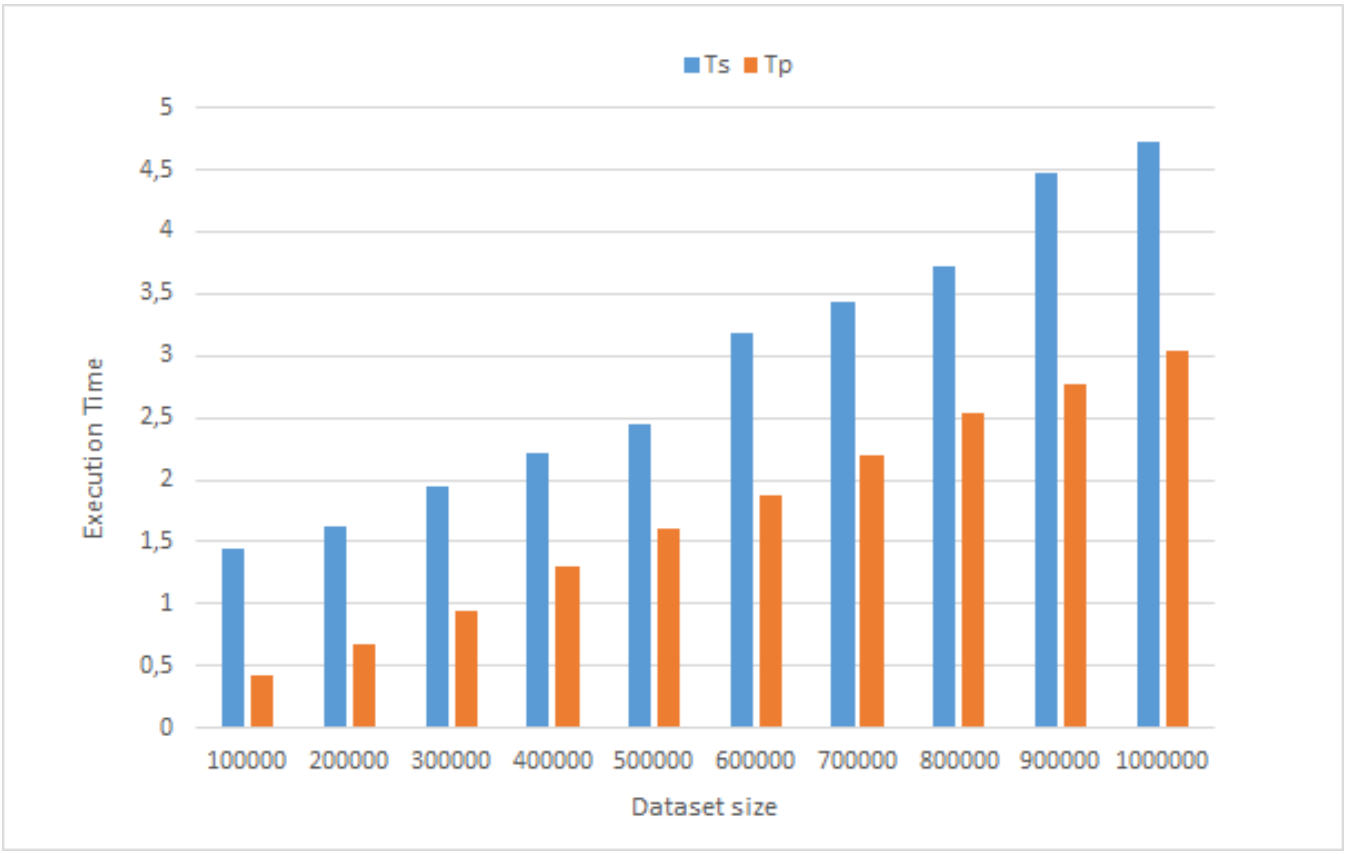
Come detto nel capitolo precedente, sono stati creati 20 differenti dataset (dieci per i dati e dieci per le etichette) ognuno con diverse dimensioni, saranno quindi utilizzati dataset con dimensione crescente da 100000 fino a 1000000 punti. All'aumentare del numero dei punti aumenterà anche il valore di k. Pertanto

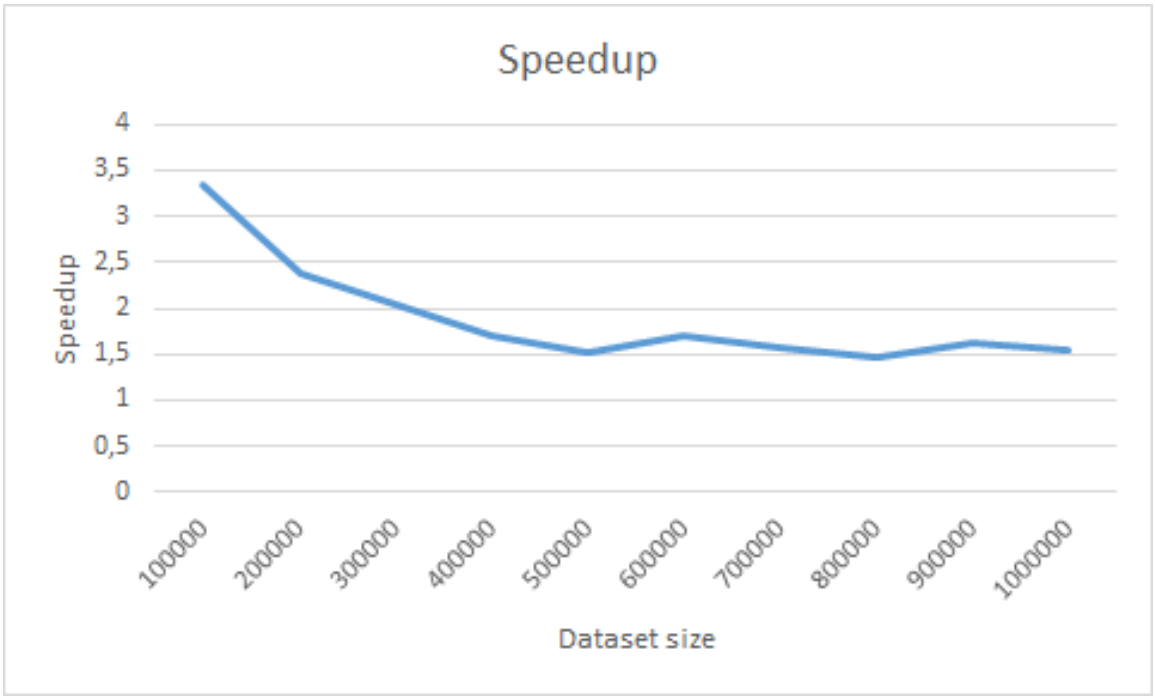
La velocità di calcolo del programma seriale viene messa a confronto con quella del programma parallelo. Viene calcolata l'accelerazione secondo la seguente equazione:

$$Speedup = \frac{T_s}{T_p} \quad —$$

Dove T_s rappresenta il tempo di esecuzione del codice seriale, T_p rappresenta quello del codice parallelo, T_d rappresenta la differenza tra i due tempi. I tempi sono rappresentati in secondi.

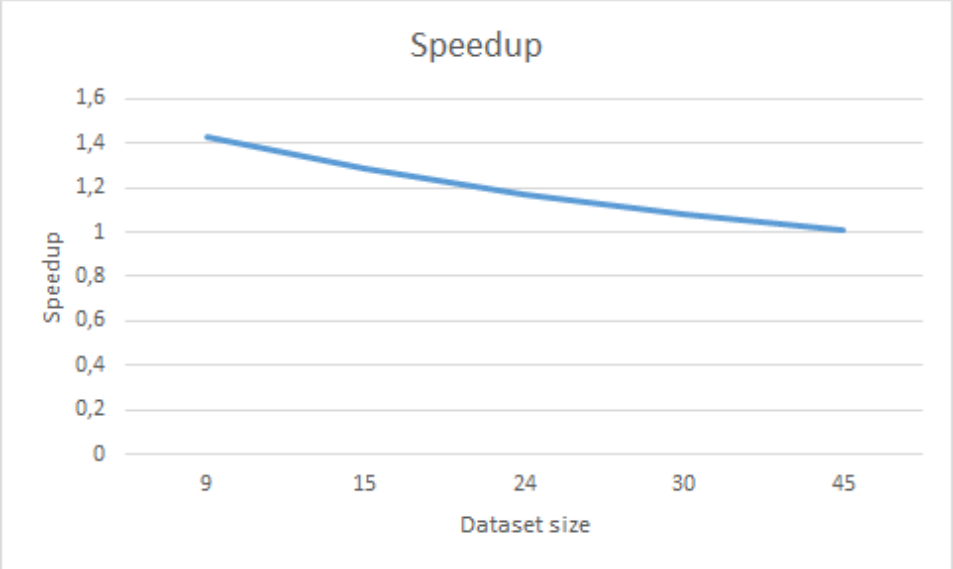
Dataset size	T_s	T_p	T_d	Speedup
100000	**1.44**	**0.43**	**1.01**	**3.34**
200000	**1.63**	**0.68**	**0.95**	**2.39**
300000	**1.95**	**0.95**	**1.00**	**2.05**
400000	**2.22**	**1.31**	**0.91**	**1.69**
500000	**2.45**	**1.61**	**0.84**	**1.52**
600000	**3.18**	**1.87**	**1.31**	**1.70**
700000	**3.44**	**2.19**	**1.25**	**1.57**
800000	**3.73**	**2.53**	**1.20**	**1.47**
900000	**4.48**	**2.77**	**1.71**	**1.61**
1000000	**4.72**	**3.04**	**1.68**	**1.55**


Ts
Tp
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Execution Time
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000
Dataset size


Speedup
4
3,5
3
2,5
2
1,5
1
0,5
0
Speedup
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000
Dataset size

Per completezza abbiamo testato il programma con un numero crescente di processi con una dimensione del dataset fissa a 500000 punti.

N_Processors	T_p	Speedup
9	2.30	1.43
15	4.50	1.29
24	6.55	1.17
30	7.80	1.08
45	11.39	1.01



Come si evince dalle misurazioni, la parallelizzazione sembra migliorare di poco le prestazioni rispetto ad un implementazione seriale. Ciò è dovuto al contesto federato che, ha obbligato l'assunzione del chunk del dataset per ogni processo tramite l'apertura del file da parte di ogni task. Tali operazioni di I/O ripetute per ogni processo, aggiunte al tempo necessario per il cambio di contesto dei thread, gravano notevolmente sull'efficienza della parallelizzazione. Per maggior chiarezza del contesto vengono riportate le misurazioni ottenute eseguendo il codice non federato.

L' efficienza si calcola con la seguente formula:

$$E = \frac{S}{p}$$

Efficienza
0,66
0,56
0,45
0,39
0,34
0,39
0,34
0,34
0,37
0,36