

**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 Dario Miligi**

Matricola n° 556126 Matricola n° 552033

**ANNO ACCADEMICO 2023/2024**

**INDICE**

**CAPITOLO 1 – INTRODUZIONE**

**CAPITOLO 2 – ARCHITETTURA E TOOLS USATI**

* 1. **MPI**

**2.2 PTHREAD**

**CAPITOLO 3 – FEDERATED LEARNING**

**CAPITOLO 4 – ALGORITMO KNN**

**4.1 ESEMPIO**

**CAPITOLO 5 – IMPLEMENTAZIONE**

**5.1 IMPLEMENTAZIONE SERIALE**

* 1. **IMPLEMENTAZIONE PARALLELA FEDERATA**

**CAPITOLO 6 – CASO DI STUDIO**

**CAPITOLO 7 – RISULTATI SPERIMENTALI**

**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.

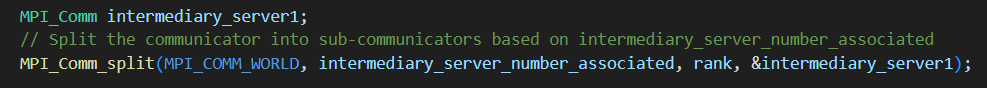
MPI utilizza oggetti chiamati comunicatori e gruppi per definire quali insiemi di processi possono comunicare tra loro. La maggior parte delle routine MPI richiede di specificare un comunicatore come argomento. MPI\_COMM\_WORLD viene utilizzato ogni volta che è richiesto un comunicatore: è il comunicatore 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:

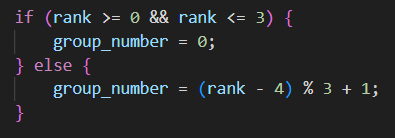
* Gruppo 0 formato dal server centrale ed i server intermedi.
* Gruppo 1 gestito dal server intermedio 1.
* Gruppo 2 gestito dal server intermedio 2.
* Gruppo 3 gestito dal server intermedio 3.

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

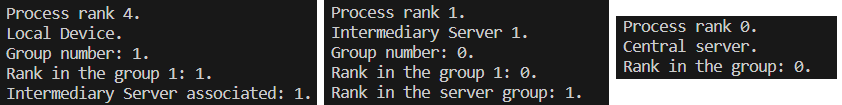
che permette di dividere il comunicatore in un secondo comunicatore. I parametri di tale funzione sono:

* 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;

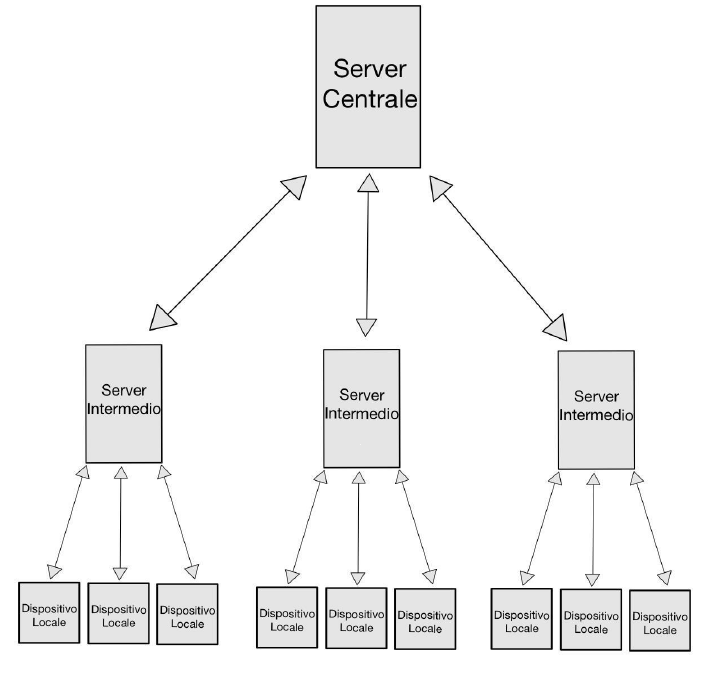
Tale istruzione viene ripetuta per ogni gruppo. Ogni processo all’interno del gruppo ottiene un nuovo rank. La suddivisione dei processi nei vari gruppi viene fatta grazie al codice della funzione determineGroupNumber(rank) che prende in ingresso il rank del processo e gli assegna 0 se il processo ha 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.



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.



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:



* 1. **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.

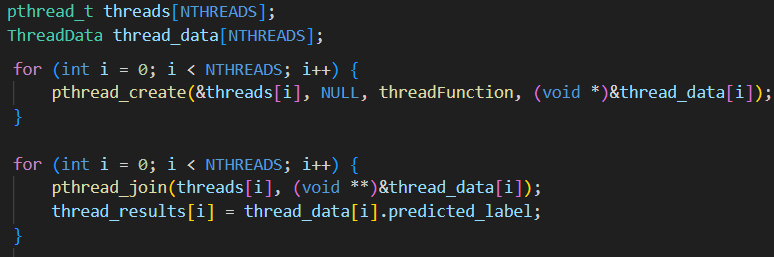
I vantaggi della programmazione mutithreading sono:

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

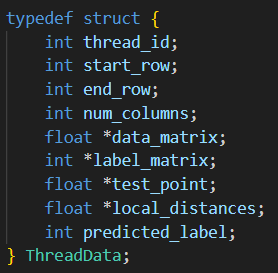
**Responsività**: L'utilizzo di thread separati consente a un'applicazione di rimanere reattiva durante operazioni lunghe o di rete.

**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.



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 dell’implementazione dell’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:

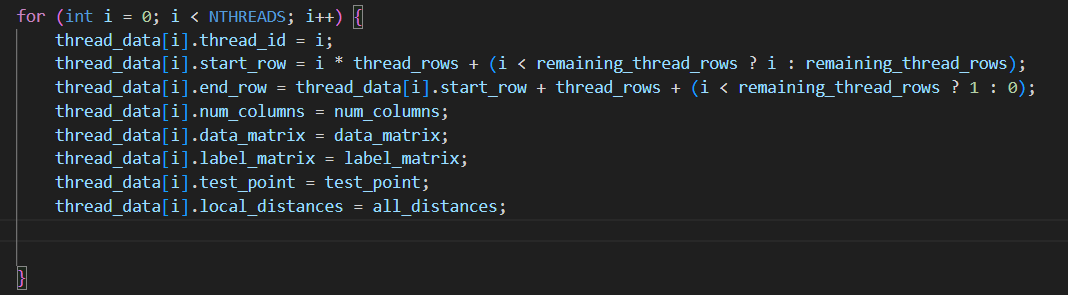
* Thread\_id: numero intero che definisce l’identificativo del thread;
* Start\_row: numero intero che definisce il numero della riga di partenza del dominio di righe del thread;
* End\_row: numero intero che definisce il numero della riga finale del dominio di righe del thread;
* Num\_columns: numero 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;
* Predicted\_label: numero intero che rappresenta la classe predetta dal thread;

I thread dunque svolgeranno la parte computazionalmente più alta del progetto, ovvero 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 alla matrice di dati del proprio processo padre, pertanto, tutti i thread accederanno a tale risorsa 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 sulla stessa matrice. 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 alla matrice del processo, questi non andranno in conflitto durante l’operazione di lettura perché andranno a operare su punti diversi della matrice.



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**

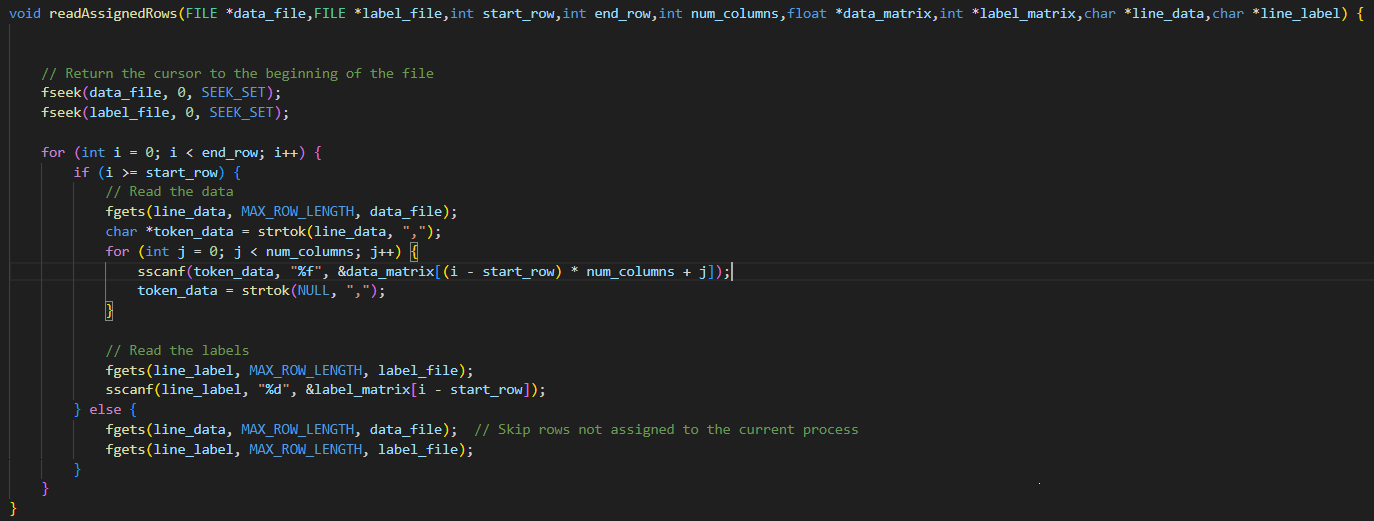
Il federated learning è un approccio all'apprendimento automatico decentralizzato e distribuito, progettato per allenare modelli di machine learning senza dover inviare dati sensibili a un server centrale. In un contesto tradizionale di apprendimento automatico, i dati vengono raccolti e inviati a un 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.

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.

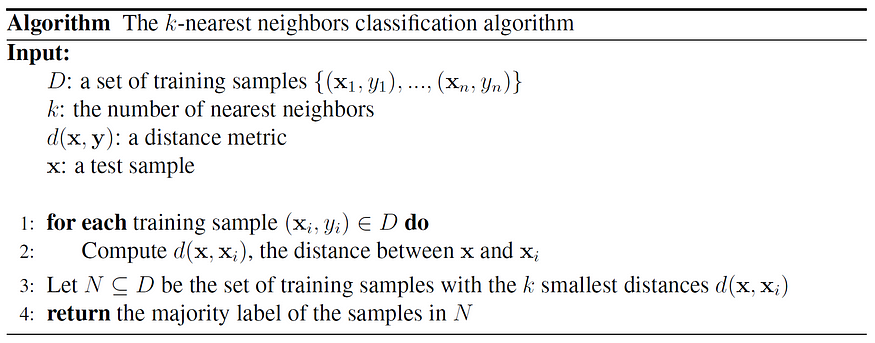
**CAPITOLO 4**

**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.

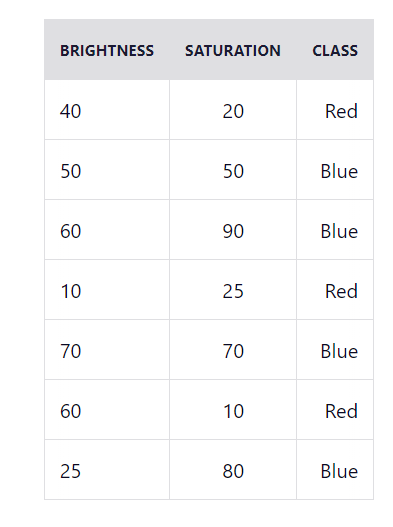
Il funzionamento di KNN può essere suddiviso in tre passaggi principali:

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.



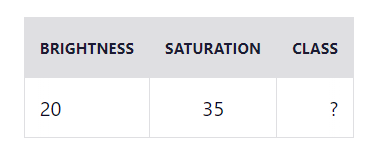
Il KNN è noto per il suo costo computazionale elevato, poiché richiede il calcolo delle distanze per ogni punto del dataset. La complessità computazionale è O(N), dove N rappresenta il numero totale di punti nel dataset. Questo rende KNN meno efficiente in scenari in cui il tempo di esecuzione è critico,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

* 1. **ESEMPIO**

Dato il seguente dataset:

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

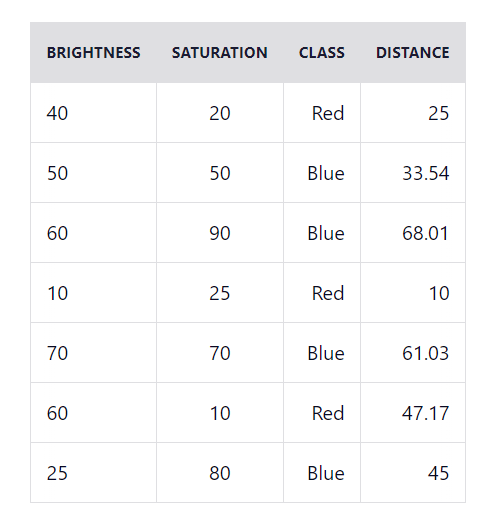
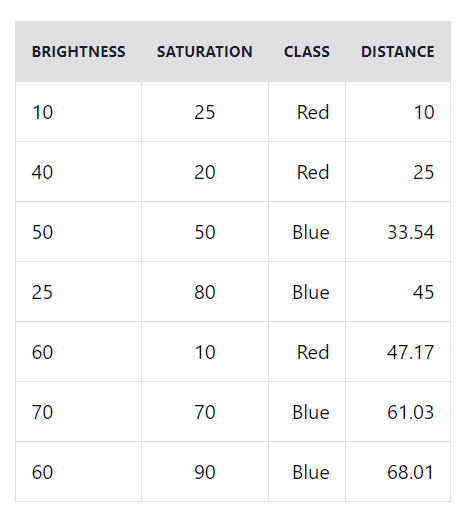
Prima di introdurre il nuovo punto da classificare, supponiamo che il valore di K sia 5.

Il punto di test sarà:

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:

Dove:

* X2 : Luminosità del punto da classificare;
* X1 : Luminosità del punto del dataset;
* Y2 : Saturazione del punto da classificare;
* Y1 : Saturazione del punto del dataset;

  
Dopo aver calcolato le distanze si ottiene la seguente tabella:  
  
Si riordinano le distanze e si ottiene:

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

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.

* 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 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.

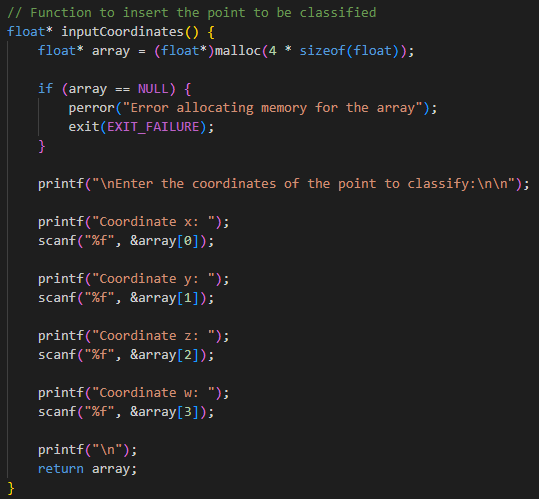
**Inizializzazione dei Dati di Addestramento**:

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.

**Stampa delle Matrici**:

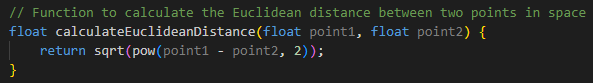
La funzione printMatrices visualizza le matrici di dati e etichette, offrendo un'opzione di debug per verificare la corretta lettura dei dati.

**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.

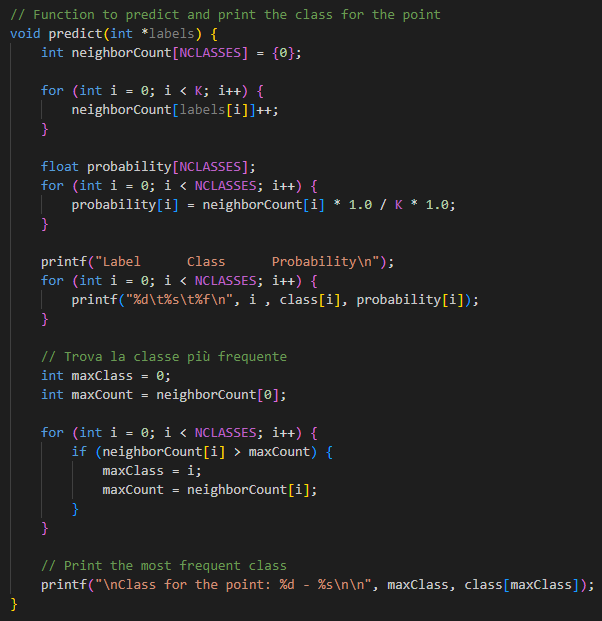
**Calcolo della Distanza Euclidea**:

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



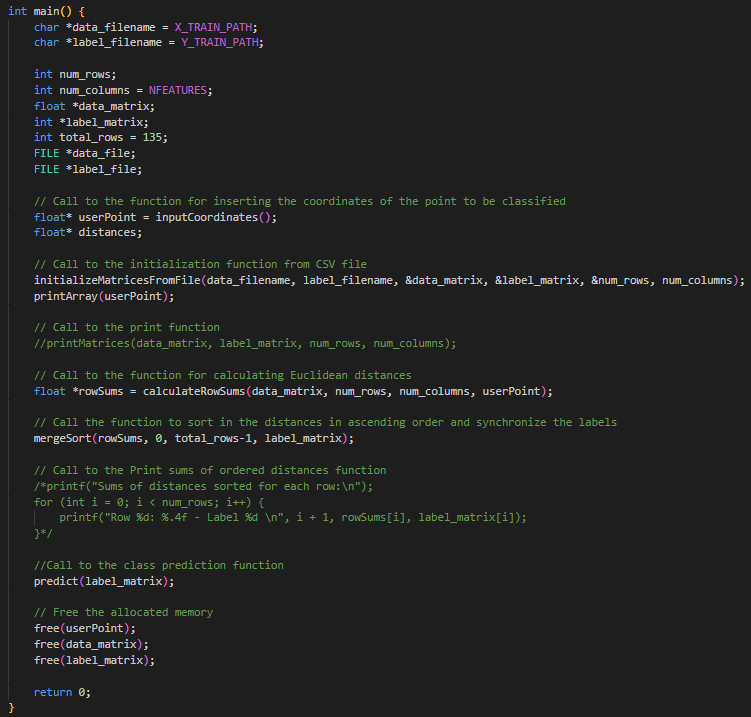
**Calcolo delle Somme delle Distanze per Ogni Riga**:

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

**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.  
  
**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.

**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.



* 1. **IMPLEMENTAZIONE PARALLELA FEDERATA**

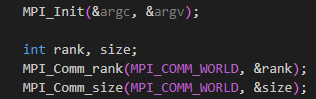
L'implementazione parallela del sistema di classificazione KNN federato si basa sulla convergenza di due paradigmi di programmazione parallela: MPI (Message Passing Interface) e Pthreads (POSIX Threads). Questa combinazione consente di sfruttare appieno le capacità di calcolo distribuito e la parallelizzazione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

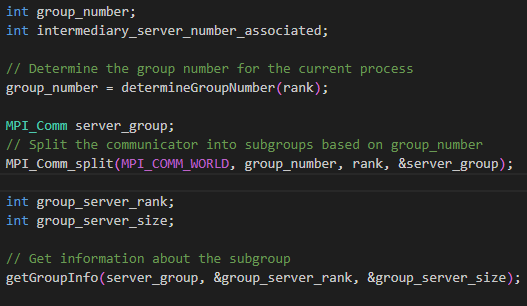
Pthreads è utilizzato per la parallelizzazione fine dei thread, permettendo di sfruttare le capacità di calcolo locali di ciascun nodo. Riprendendo anche in questo caso i concetti del capitolo 2.2, ogni processo MPI crea un numero specificato di thread Pthreads per gestire la computazione parallela dei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

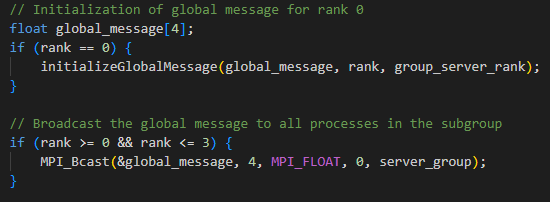
L'aspetto federato del sistema si traduce nella decentralizzazione della gestione del dataset. Ogni nodo di calcolo è autonomo nella gestione del proprio "chunk" di dati, consentendo una maggiore efficienza e riducendo la dipendenza da una gestione centralizzata e soprattutto implementando la privacy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

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.

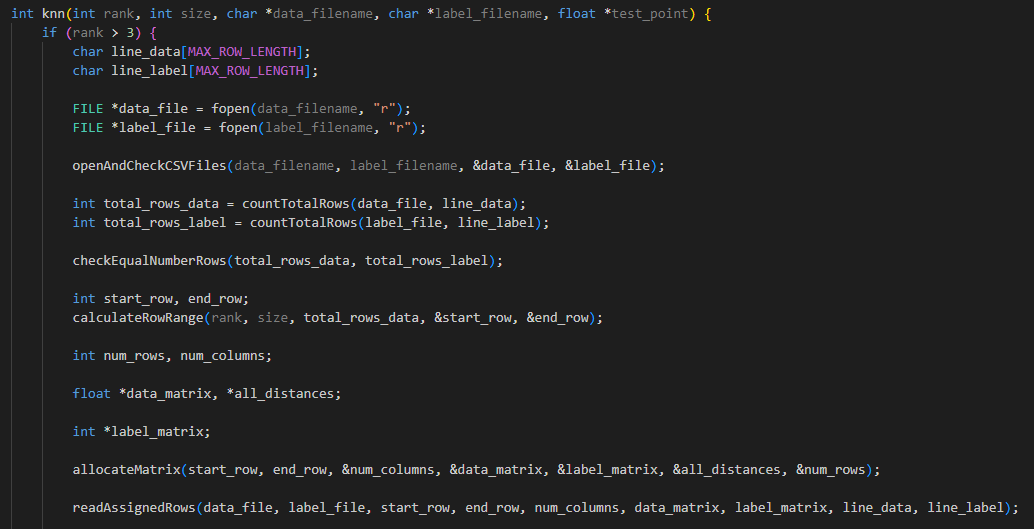


Si procede poi con l’inizializzazione delle coordinate per il punto da classificare. Questa operazione viene effettuata soltanto dal server centrale, che si occuperà di inviare in broadcast (utilizzando la primitiva MPI\_Bcast) il punto inserito dall’utente ai vari server intermedi tramite il comunicatore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

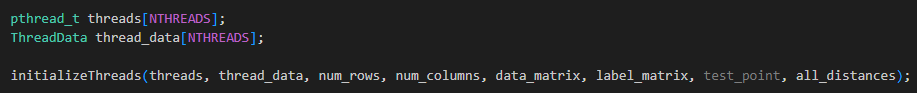


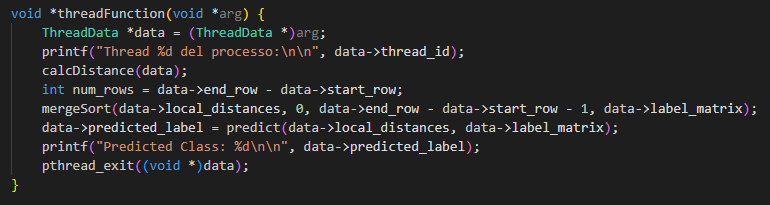
Segue poi la suddivisione dei processi nei restanti gruppi con conseguente invio del punto da classificare in broadcast da parte del server intermedio per il proprio gruppo. L’operazione è stata eseguita con la medesima strategia per tutti i gruppi.

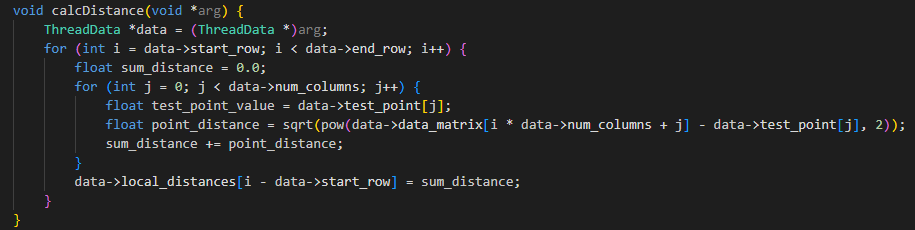
A questo punto in cui ogni processo ha il proprio ruolo all’interno della struttura gerarchica ed ha conoscenza sul punto da classificare, è necessario assumere per ogni processo la propria porzione di dati di addestramento e di etichette provenienti dai file CSV. Il flusso di lavoro per questa determinata 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.



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.

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 l’algoritmo knn sulla propria matrice di dati ed etichette.

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 knn. Tali estremi saranno utili nella funzione che calcola le distanze euclidee tra i punti appartenenti al dominio del thread.

Tale funzione calcola la distanza euclidea secondo la formula menzionata nel capitolo 4.1.

Inizio modulo