

**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 – CONTESTO FEDERATO**

**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.

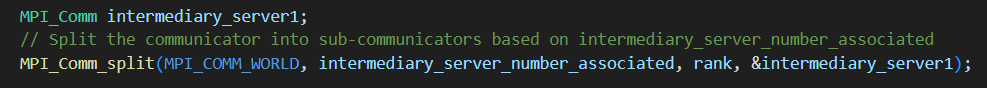
MPI utilizza oggetti chiamati comunicatori e gruppi per definire quali insiemi di processi possono comunicare tra loro. La maggior parte delle routine MPI richiede di specificare un comunicatore come argomento. MPI\_COMM\_WORLD viene utilizzato ogni volta che è richiesto un comunicatore: è il comunicatore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

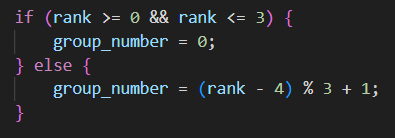
* Gruppo 0 formato dal server centrale ed i server intermedi.
* Gruppo 1 gestito dal server intermedio 1.
* Gruppo 2 gestito dal server intermedio 2.
* Gruppo 3 gestito dal server intermedio 3.

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

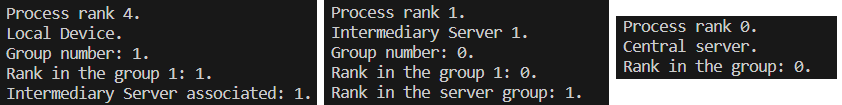
che permette di dividere il comunicatore in un secondo comunicatore. I parametri di tale funzione sono:

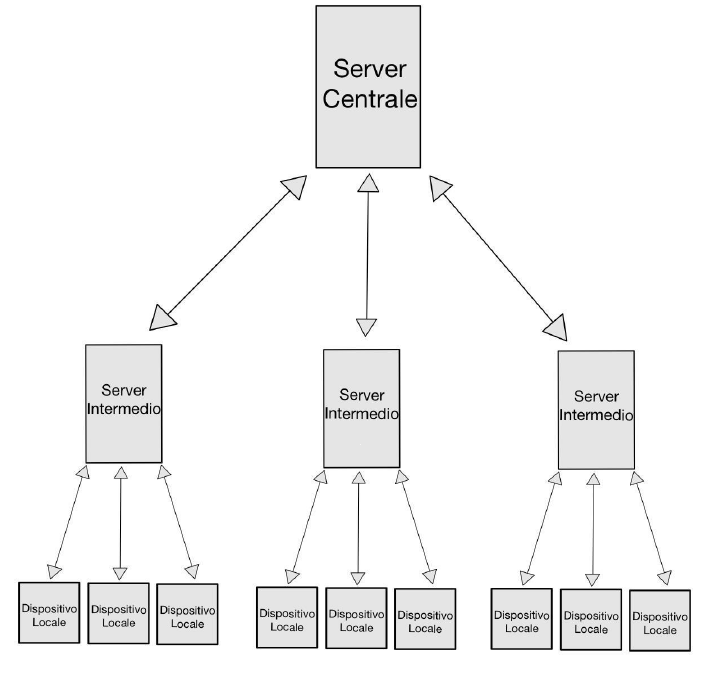
* MPI\_COMM\_WORLD: comunicatore da dividere;
* Intermediary\_server\_number\_associated: rappresenta un tag per identificare il comunicatore, tale tag sarà utile per assegnare i processi al gruppo (comunicatore);
* Rank: rango del processo;
* &intermediary\_server: puntatore al nuovo comunicatore;

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:

**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:

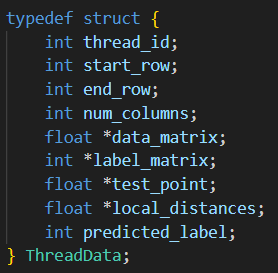
**Parallelismo**: I thread possono eseguire operazioni in parallelo, sfruttando i processori multi-core per migliorare le prestazioni.

**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.

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 thread;
* \*data\_matrix: puntatore alla matrice locale di dati float del thread;
* \*label\_matrix: puntatore alla matrice locale di etichette intere del thread;
* \*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.