Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Progettazione e sviluppo di una versione distribuita di un algoritmo di subspace clustering

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:
Dott. Moreno Marzolla

Presentata da:
Marco Verrocchio

Correlatore:
Dott. Matteo Magnani

Sessione III
Anno Accademico 2011-2012

Dedicato alla comunità dei ricercatori.

Indice

Introduzione

Il task del data mining si pone come obiettivo l'estrazione automatica di schemi significativi da grandi quantità di dati.
Un esempio di schemi che possono essere cercati sono raggruppamenti significativi dei dati, si parla in questo caso di **clustering**.
Gli algoritmi di clustering tradizionali mostrano grossi limiti in caso di dataset ad alta dimensionalità, composti cioè da oggetti descritti da un numero consistente di attributi. Di fronte a queste tipologie di dataset è necessario quindi adottare una diversa metodologia di analisi: il **subspace clustering**.
Il subspace clustering consiste nella visita del reticolo di tutti i possibili sottospazi alla ricerca di gruppi significativi (cluster)
Una ricerca di questo tipo è un'operazione particolarmente costosa dal punto di vista computazionale.
Diverse ottimizzazioni sono state proposte al fine di rendere gli algoritmi di subspace clustering più efficienti [1].

In questo lavoro di tesi si è affrontato il problema da un punto di vista diverso: l'utilizzo della parallelizzazione al fine di ridurre il costo computazionale di un algoritmo di subspace clustering: **Inscy** [2].

Il lavoro di tesi si è articolato dapprima in uno studio approfondito dell'articolo di INSCY allo scopo di comprenderne a fondo la teoria di base.
Non esistendo una implementazione di riferimento di INSCY si è reso necessario dapprima realizzarne un'implementazione sequenziale basandosi unicamente sulla teoria studiata nell'articolo.
Successivamente, partendo dalla versione sequenziale implementata, si è progettata e implementata la parte di parallelizzazione; a tal fine è stato utilizzato MPI e in particolare la libreria OpenMPI e il wrapper boost/mpi.
Infine sono stati eseguiti alcuni test allo scopo di studiare la scalabilità dell'algoritmo implementato.

L'algoritmo è stato chiamato **dINSCY (distributed INSCY)**.
Questo elaborato è cosi organizzato:

- Nel capitolo 1 viene fatta una breve introduzione al task del **clustering**: vengono definiti i concetti base e data una classificazione delle principali tecniche di clustering
- Nel capitolo 2 viene spiegato più in dettaglio il **subspace clustering**: se ne mostra dapprima il contesto di utilizzo per poi classificare ed elencare i principali algoritmi di subspace clustering. Infine viene fornita una spiegazione dettagliata (dal punto di vista teorico e con un esempio pratico di esecuzione) del funzionamento di INSCY.
- Nel capitolo 3 viene fatta una breve introduzione al calcolo parallelo
- Nel capitolo 4 viene mostrata e spiegata nel dettaglio l'implementazione di dINSCY
- Infine nel capitolo 5 vengono mostrati e discussi i risultati dei test di scalabilità di dINSCY
- Inoltre nell'appendice A viene fatta una breve introduzione allo standard MPI

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione al task del clustering

In questo capitolo verrà introdotto il concetto clustering dal punto di vista teorico. Dapprima verrà data una definizione del contesto in cui il clustering va a inserirsi, il Data Mining, per poi passare alla definizione e alla spiegazione delle diverse tecniche di clustering dando particolare enfasi a quelle alla base dell'algoritmo sviluppato per questo lavoro di tesi.

1.1 Data Mining

Il Data Mining [5] è un insieme di tecniche e metodologie che hanno lo scopo di estrarre informazioni utili da grandi quantità di dati. Il Data Mining costituisce la parte di modellazione del processo di **Knowledge Discovery in Databases** ovvero l'intero processo di estrazione di informazione dai dati, che va dalla selezione e pre-processamento dei dati fino all'interpretazione e valutazione del modello ottenuto dal processo di Data Mining.

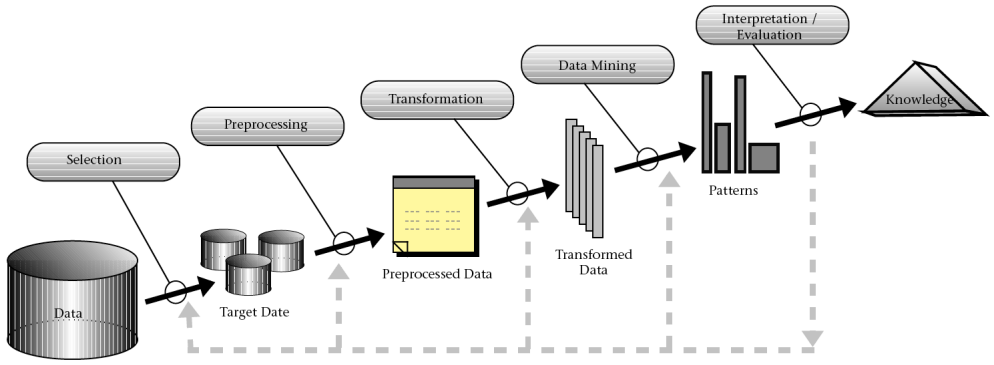


Figura 1.1: KDD [3]

In figura 1.1 viene mostrato il processo di *Knowledge Discovery*; questo si suddivide in 6 fasi, le quali hanno una serie di dati in input e restituiscono in output i dati processati:

1. Selezione: in questa fase viene creato il dataset obiettivo attraverso la selezione dai dati d'origine di un sottoinsieme di dati su cui eseguire il processo di discovery

2. Pre-processamento: in questa fase vengono eseguite diverse operazioni atte alla "pulizia" del dataset obiettivo come ad esempio la rimozione di eventuali dati non rilevanti (o rumore) o la decisione della strategia da utilizzare per gestire eventuali dati sparsi

3. Trasformazione: in questa fase vengono selezionati gli attributi rilevanti per rappresentare il dataset; questi dipendono dall'obiettivo che si vuole raggiungere e sono usate tecniche di riduzione della dimensionalità

4. Data-mining: questa è la fase in cui sul dataset (pre-processato nelle fasi precedenti) vengono ricercati pattern significativi

5. Interpretazione/Valutazione: in questa fase si cerca di interpretare i pattern ottenuti dalla fase di data-mining per dare loro un significato

Parlando più nel dettaglio della fase 4 del KDD, esistono diverse tecniche di data-mining: queste si suddividono in **supervisionate**, in cui l'algoritmo

deve apprendere il comportamento di alcune variabili target[1], e **non supervisionate** in cui le variabili target non sono conosciute a priori. Tra le tecniche supervisionate troviamo:

- Alberi decisionali
- Reti bayesiane
- Reti neurali

Tra le tecniche non supervisionate troviamo invece il **clustering** e le regole associative, parleremo ora più in dettaglio della prima.

1.2 Il task del clustering

Il clustering [6] è una tecnica di analisi che ha lo scopo di individuare gruppi significativi in un dataset. La parola "significativi" in questo contesto ha un'accezione puramente topologica: un cluster viene definito come "un insieme di oggetti tale che la similitudine tra gli oggetti all'interno del cluster sia maggiore della similitudine tra oggetti interni e quelli esterni al cluster" [6].

Gli algoritmi di clustering possono essere suddivisi secondo le relazioni tra i cluster trovati:

- **partitivi**: trovano cluster disgiunti, ovvero nei quali ogni oggetto appartiene esclusivamente ad un unico cluster Questi algoritmi possono essere a loro volta suddivisi a seconda del criterio di clustering:
  - **rilocativi**: cercano di trovare i cluster riposizionando gli oggetti iterativamente
  - **probabilistici**: utilizzano distribuzioni probabilistiche per modellare i cluster (es. distribuzioni normali multivariate per l'algoritmo EM)
  - **basati su centroidi**: utilizzano degli oggetti rappresentativi per ogni cluster (chiamati centroidi)
  - **basati su densità**: cercano di individuare regioni dense dello spazio, questi algoritmi sono particolarmente indicati per i cluster di forma convessa

[1] variabili il cui valore non è noto ma viene predetto in funzione di altre variabili già conosciute dette predittive

- **gerarchici**: trovano insieme di cluster annidati organizzati ad albero (dendogramma), ogni oggetto in questo caso può appartenere contemporaneamente a più cluster nella gerarchia.

  Questi possono essere a loro volta suddivisi a seconda della strategia di costruzione:

  · **agglomerativi**: utilizzano una strategia bottom-up, inizialmente ogni oggetto è un cluster e si procede via via unendo i cluster

  · **divisivi**: utilizzano una strategia top-down, inizialmente l'intero dataset forma un unico cluster e si procede via via dividendo in cluster diversi.

- **Grid-based**: non utilizzano direttamente i dati ma effettuano una quantizzazione dello spazio in regioni e lavorano sulle regioni invece che sui dati

- **metodi basati sulla co-occorrenza di dati categoriali**

- **clustering di dataset ad alta dimensionalità o subspace clustering**: algoritmi nei quali i cluster trovati possono essere sovrapposti ma risiedono in sottospazi diversi.

1.3 DBSCAN

DBSCAN [7, 4] è un algoritmo di clustering basato su densità, in particolare fa parte dell'insieme degli algoritmi basati su connettività ovvero che computano la densità sui singoli oggetti del dataset. L'altra categoria di algoritmi basati su densità sono quelli *density-function* che invece cercano di calcolare una funzione di densità sull'intero dataset.

Di seguito alcune definizioni usate da DBSCAN:

ϵ-neighborhood di un oggetto p: Sia $d(p, q)$ la distanza euclidea tra due oggetti p e q, definiamo con $N_\epsilon(p)$ il numero di oggetti posizionati entro un raggio ϵ da p
$N_\epsilon(p) = \{q \in D | d(p, q) \leq \epsilon\}$

direct density-reachability:
sia MinPts un parametro in input che indica il numero minimo di oggetti

che un cluster deve contenere.
Un oggetto p è directly density-reachable da un punto q rispetto a ϵ se q è a una distanza inferiore di ϵ da p e esistono abbastanza punti nell'ϵ vicinato di p
$q \in N_\epsilon(p)$
$|N_\epsilon(p)| \geq$ MinPts

density-reachability: un oggetto p è density-reachable da un punto q rispetto a ϵ e MinPts se esiste una catena di punti direct density-reachable l'uno dall'altro che collegano p a q:
$\exists p_1, \cdots, p_n, p_1 = q, p_n = p$ t.c. p_{i+1} è directly density reachable da p_i Questa relazione non è simmetrica, infatti potremmo avere che q non abbia abbastanza punti da essere considerato denso e quindi varrebbe la relazione da p a q ma non da q a p, si va quindi a introdurre il concetto di density-connectivity

density-connectivity: un oggetto p è density-connected con un punto q rispetto a ϵ e MinPts se esiste un punto o che sia density-reachable da entrambi:
$\exists o$ t.c. p è density-reachable da o e q è density-reachable da o
Questa relazione è simmetrica.

density-based cluster: dato un dataset D, un density-based cluster C è un sottoinsieme non vuoto di D che soddisfi le seguenti condizioni:

massimalità: $\forall p, q$: se $p \in C$ e q è density-reachable da $p \Rightarrow q \in C$

connettività: $\forall p, q \in C$: p è density-connected a q

rumore: si definisce rumore l'insieme dei punti che non appartengono ad alcun cluster

L'algoritmo DBSCAN inizia prendendo un oggetto arbitrario p del dataset; ne va a calcolare $\epsilon -$ neighborhood: se al suo interno vi sono più di MinPts oggetti, p viene etichettato come appartenente a un cluster, altrimenti come rumore. A questo punto si cerca di espandere il cluster appena trovato includendo iterativamente gli oggetti negli $\epsilon -$ vicinati di quelli già appartenenti al cluster, questo processo continua fino a quando il cluster non possa più essere espanso. Si passa quindi ad analizzare il successivo oggetto non visitato del dataset.

1.4 Clustering di spazi ad alta dimensionalità

Il campo di applicazione del subspace clustering [8] è quello dei dataset ad alta dimensionalità ovvero i dataset con un numero alto di attributi. Questi, a differenza dei dataset che si sviluppano su poche dimensioni, portano interessanti problemi nel campo del clustering:

- per definizione un cluster è un insieme di oggetti "simili"; la similitudine è un parametro dipendente dagli attributi che descrivono gli oggetti. In un dataset ad alta dimensionalità non sempre tutti gli attributi hanno la stessa importanza: a seconda dellà realtà che il dataset descrive alcuni attributi possono essere più importanti di altri, questa distinzione non può essere catturata dagli algoritmi classici di clustering.

- il concetto di similarità (intesa come distanza tra oggetti) tende a perdere di significato all'aumentare del numero delle dimensioni, infatti la distanza tra ogni coppia di punti tende a diventare 0 all'aumentare del numero di dimensioni. Questo problema è definito **maledizione della dimensionalità**.

Il problema di individuazione degli attributi rilevanti ricade nel campo del preprocessamento del dataset: si utilizzando una serie di tecniche di *feature selection*.

Per il problema della maledizione della dimensionalità si utilizzano principalmente due tecniche per la riduzione della dimensionalità del dataset:

- **trasformazione degli attributi**: vengono selezionati gli attributi più importanti e solo su quelli viene effettuato il clustering

- **decomposizione del dominio**: il dataset viene diviso in tanti piccoli sottoinsiemi facendo in modo cosi che l'algoritmo di clustering venga eseguito si su tutte le dimensioni ma su un "mini" dastaset.

Gli algoritmi di subspace clustering cercano di superare i problemi di cui sopra andando a cercare i cluster candidati in tutti i sottospazi possibili dello spazio iniziale; una volta trovati questi vengono passati a un algoritmo di clustering tradizionale (es. DBSCAN) che trova eventuali cluster nel dataset ristretto.

Capitolo 2

Lo stato dell'arte

In questo capitolo analizzeremo più in dettaglio il subspace clustering cercando dapprima di comprenderne meglio le motivazioni per poi passare a presentare alcuni algoritmi che ne rappresentano lo stato dell'arte.

Come già detto, due sono i problemi principali quando si vanno a trattare dataset ad alta dimensionalità, **presenza di dimensioni irrilevanti** e **la maledizione della dimensionalità**.

Vengono introdotte due tecniche di *riduzione della dimensionalità* per cercare di risolvere questi problemi:

> **feature transformation**: con questa tecnica si cerca di ridurre la dimensionalità mappando lo spazio originario in uno spazio di più bassa dimensionalità che ne è una trasformazione lineare. Un esempio di feature transformation è la *Principal Component Analysis* o *PCA* [9] e la *Singular Value Decomposition* o *SVD* [10].
>
> **feature selection**: con questa tecnica si riduce la dimensionalità mappando lo spazio originario in uno spazio che è ottenuto selezionandone alcune dimensioni (quelle supposte più rilevanti).

Queste tecniche di riduzione della dimensionalità presentano tre problemi, in primo luogo mentre gli attributi originari possono avere un qualche significato, le loro trasformazioni/selezioni perdono di significato e cosi i cluster che vengono trovati non sono intuitivamente comprensibili. In secondo luogo queste tecniche riducono sì la dimensionalità ma il risultato è che un successivo algoritmo di clustering troverà i cluster solo nello spazio ridotto non considerando tutti gli

altri possibili sottospazi e questo potrebbe portare a perdere informazioni sugli oggetti che appartengono a più cluster in differenti sottospazi.

Un secondo approccio per trattare i dataset ad alta dimensionalità è il **projected clustering**. Con questa tecnica non è più un singolo sottospazio ad essere considerato ma delle coppie $(C_i, S_i)_{(0 \leq i \leq k)}$ con C_i i-esimo cluster e S_i sottospazio in cui esiste C_i. Il problema di questo approccio è che si continua a non riuscire a catturare le informazioni sugli oggetti appartenenti a diversi cluster su differenti sottospazi.

Con il subspace clustering si cerca di risolvere questa problematica attraverso la rilevazione automatica di cluster in tutti i sottospazi dello spazio originale.

Verranno ora presentati alcuni algoritmi di subspace clustering

2.1 Algoritmi grid-based

Uno dei primi algoritmi di subspace clustering è stato *Clique* (*CLustering in QUEst*) [10]: questo è un algoritmo bottom-up grid-based che usa un metodo in stile *apriori* per navigare tutti i possibili sottospazi.
L'algoritmo procede come segue:

1. Ogni dimensione dello spazio viene partizionata in blocchi equi-dimensionati di larghezza ϵ (parametro di input dell'algoritmo); questi blocchi sono chiamati **unità**.

2. viene calcolata la densità di oggetti in ogni unità e solo le unità con densità che supera una soglia τ (input dell'algoritmo) vengono mantenute.

3. l'algoritmo procede quindi in stile bottom-up: la generazione delle unità k-dimensionali viene fatto a partire dalle unità $(k-1)$-dimensionali unendo quelle che hanno le prime $(k-2)$ dimensioni in comune, dalle unità cosi generate vengono eliminate quelle non dense.

4. Viene applicato un criterio di pruning con il quale si vanno a tagliare quei sottospazi con bassa copertura (frazione di punti nel sottospazio rispetto ai punti totali del dataset).

5. Si procede trovando i cluster, questi sono definiti come insiemi massimali di unità dense connesse. In ogni sottospazio k-dimensionale viene computato un insieme disgiunto di unità connesse k-dimensionali.

6. Infine viene generata per ogni insieme la Minimal Cluster Description: ogni insieme computato viene coperto da regioni massimali e successivamente ne viene determinato la copertura minima.

Un algoritmo che è una evoluzione di CLIQUE è **MAFIA** (Merging of Adaptive Finite IntervAls) [11], questo è sempre un algoritmo grid-based ma invece di usare una griglia con unità di larghezza fissata utilizza una griglia adattiva di larghezza variabile per ogni dimensione. Per MAFIA il concetto di unità densa è dato da un criterio chiamato *cluster dominance factor*: vengono mantenute solo quelle unità che siano α volte (con α parametro di input) più densamente popolate della media. Un'altra importante differenza tra CLIQUE e MAFIA è che quest'ultimo nella sua strategia bottom-up trova le unità k-dimensionali partendo da quelle $(k-1)$-dimensionali unendo non solo quelle che hanno le prime $(k-2)$ dimensioni in comune ma tutte quelle che hanno qualunque $(k-2)$ dimensioni in comune. Questo porta ad avere un numero di unità k-dimensionali generate molto più alto.
Successivamente, come in CLIQUE, le unità dense vicine vengono unite per formare i cluster e i cluster ridondanti (rispetto a cluster trovati in uno spazio a più alta dimensionalità) vengono tagliati.

Una grande limitazione di questi algoritmi è data proprio dall'utilizzo di una griglia multi-dimensionale: i cluster sono confinati all'interno delle celle della griglia. Ad esempio unità k-dimensionali vicine che singolarmente non sono dense potrebbero diventare unità dense se unite, svelando un eventuale cluster. Gli algoritmi presentati finora non tengono in considerazione questa eventualità con la possibilità quindi di non trovare cluster presenti nel dataset.

2.2 Algoritmi density-based

L'algoritmo **SUBCLU** [12] supera le limitazioni di cui sopra eliminando il concetto di griglia, andando invece a utilizzare il concetto di regione densa e di cluster utilizzato da *DBSCAN*.

SUBCLU, similmente agli algoritmi presentati, usa una strategia bottom-up per generare i sottospazi. Inizia applicando DBSCAN allo scopo di trovare i cluster in tutti gli spazi 1-dimensionali. Successivamente per ogni cluster generato viene controllato se sia totalmente (o parzialmente) coperto da qualche altro clu-

ster in un sovraspazio dello spazio attuale. A questo punto, per ogni sottospazio k-dimensionale $S \in S_k$, cerchiamo tutti gli altri sottospazi $T \in S_k$ con $T \neq S$ aventi $(k-1)$ attributi in comune e li uniamo per generare i sottospazi $(k+1)$-dimensionali.
Per la proprietà di monotonicità, per ogni sottospazio $S \in S_{k+1}$ cosi generato S_k deve contenere ogni sottospazio k-dimensionale T tale che $T \subset S$. Possiamo quindi tagliare quei candidati $(k+1)$-dimensionali che hanno almeno un sottospazio k-dimensionale non incluso in S_k.
Nell'ultimo passo l'algoritmo va a generare i cluster $(k+1)$-dimensionali e i corrispondenti sottospazi $(k+1)$-dimensionali che contengono questi cluster, cioè per ogni sottospazio S $(k+1)$-dimensionale si va a prendere il sottospazio k-dimensionale T tale che $T \subset S$ e su ogni cluster in T viene eseguito DBSCAN.
Per minimizzare il costo di DBSCAN si sceglie il sottospazio T_b che contenga il numero minore di oggetti nei cluster in esso contenuti.

$$T_b = \min_{s \in S_k \wedge s \subseteq S} \sum_{C_i \in C^s} |C_i|$$

con s sottospazio, C_i i-esimo cluster di s e C^s insieme dei cluster di s.
Grazie a questa euristica vengono minimizzate il numero di computazioni del vicinato effettuate da DBSCAN riducendo di molto il costo computazionale complessivo dell'algortimo.

Riassumendo, da una parte gli algoritmi grid-based mostrano un tempo di esecuzione basso ma al costo di una precisione nell'individuazione dei cluster bassa (dovuta alle euristiche introdotte e al posizionamento della griglia), dall'altra parte gli algoritmi density-based mostrano una precisione notevolmente più alta ma al costo di un tempo di esecuzione molto elevato dovuto alle continue computazioni dei vicinati (necessarie per il calcolo della densità).

Un algoritmo di subspace clustering necessita di esibire un compromesso tra precisione e tempo di esecuzione, nella prossima sezione verrà mostrato **INSCY**, un algoritmo di subspace clustering density-based che grazie a una struttura adhoc di indicizzazione dei sottospazi e ad una euristica di pruning degli stessi permette un abbassamento notevole dei tempi di esecuzione pur mantenendo un'ottima precisione.
Questo algoritmo è alla base dell'algoritmo sviluppato in questo lavoro di tesi, che ne è una sua versione distribuita.

2.3 INSCY

INSCY [2, 13] nasce da due necessità:

- trovare tutti e i soli cluster non ridondanti in tutti i sottospazi
- mantenere un tempo di esecuzione basso

Queste due necessità vengono risolte attraverso una strategia di visita del reticolo dei sottospazi di tipo depth-first: per ogni regione viene effettuato il mining di tutti i cluster in tutti i sottospazi prima di passare alla regione successiva. Questo porta due vantaggi principali, effettuare dapprima il mining degli spazi ad alta dimensionalità permette di tagliare tutti i sottospazi ridondanti; in secondo luogo con questa strategia di visita è possibile un'efficiente indicizzazione di tutti i possibili sottospazi.
INSCY utilizza per la rappresentazione del dataset una *griglia a conservazione di densità* per la discretizzazione dello spazio unita a una struttura dati ad-hoc per l'indicizzazione dello stesso: lo **SCY-tree**.
Prima di vedere più in dettaglio il funzionamento dell'algoritmo definiamo il concetto di *griglia a conservazione di densità* e di SCY-tree, daremo poi la definizione di cluster utilizzata da INSCY.
Definizione: Griglia a conservazione di densità
Una griglia a conservazione di densità è una griglia cosi come definita dagli algoritmi di clustering grid-based (ovvero una suddivisione di ogni dimensione in intervalli di larghezza fissata) con in più l'aggiunta dei **bordi di connettività** ovvero regioni di larghezza ϵ posizionate nel bordo di ogni intervallo
Definizione: SCY-tree:
Uno SCY-tree T_D rappresenza la regione $T_d = ((d_1, i_1) \cdots (d_k, i_k))$. Una regione è ottenuta partendo dallo spazio "pieno" per restrizioni successive su (d_1, i_1), (d_2, i_2), (d_3, i_3) con (d_i, i_i) che indica la coppia dimensione, intervallo su cui si restringe. Restringere lo spazio su una coppia (d, i) significa prendere in considerazione solo quegli oggetti dello spazio che nella dimensione d siano posizionati nell'intervallo i, eliminando tutti gli altri.
Uno SCY-tree consiste di una serie di nodi che contengono:

- un descrittore (d, i) con d dimensione e i intervallo
- un puntatore al nodo parent e una lista di puntatori ai figli

- un puntatore di una lista concatenata di nodi con stesso descrittore

I nodi nello SCY-tree sono ordinati lessicograficamente secondo i loro descrittori (prima sulla dimensione e poi sull'intervallo).
Rappresentiamo i nodi con la sintassi $(d, i) : c$ con c contatore degli oggetti referenziati dal nodo (d, i)
I bordi di connettività sono rappresentati attraverso speciali nodi $(d, i) : -1$ chiamati *S-connettori*, diamo al contatore il valore -1 in quanto non siamo interessati al numero degli oggetti in questa regione di bordo ma solo alla loro presenza.

Definizione: Subspace Cluster:
Sia DB il database contenente gli oggetti del dataset.
Un subspace cluster (C, S) è un insieme $C \in DB$ di oggetti con $|C| \geq minSize$ (numero minimo di oggetti che un cluster deve avere, parametro di input).
Sia o^s la proiezione dell'oggetto o nel sottospazio s, definiamo il cluster C come un cluster tale che:

- gli oggetti in C sono **s-densi**: ogni oggetto del cluster ha almeno τ_s (soglia minima di densità, parametro di input) oggetti nel suo vicinato (indicato con N^s_ϵ)
  $\forall o^s \in C \ : \ |N^s_\epsilon(o)| \geq \tau_s$ con $N^s_\epsilon(q) = \{p^s \in \text{DB} | d(p^s, q^s) \leq \epsilon\}$;

- gli oggetti in C sono **s-connessi**: tra ogni coppia di oggetti p e q contenuti nel cluster esiste una catena di oggetti a loro volta s-connessi
  $\forall o^s, p^s \in C \ : \ \exists q^s_1, \cdots, q^s_m \in C$ t.c. $q^s_1 = o^s \wedge q^s_m = p^s \wedge \forall i \in \{2, \cdots, m\}\ q^s_i \in N^S_\epsilon(q^s_{i-1})$

- C è **massimale**: non esistono oggetti esterni al cluster che siano s-connessi con oggetti interni al cluster
  $\forall o^s, p^s \in \text{DB} \ : \ $ se o^s, p^s s-connected $\Rightarrow (o^s \in C \Leftrightarrow p^s \in C)$

- (C, S) non è **ridondante**: non esiste un cluster in un sovraspazio di S sottoinsieme di C secondo un fattore di ridondanza R
  $\nexists (C', S')$ t.c. $C' \subseteq C \wedge S' \supset S \wedge |C'| \geq R * |C|$ con R fattore di ridondanza (100% piena ridondanza, 0% nessuna ridondanza)

L'ultimo concetto che ci resta da definire è l'in-process removal, Diamo la definizione di densità e poi andremo a definire la proprietà di **monotonicità della densità**
Definizione: densità debole

Sia expDen(n) la densità attesa nello spazio n-dimensionale ovvero il numero medio di oggetti in tutti gli ϵ-vicinati dello spazio.
Un oggetto $O \in$ DB è debolmente denso se

$$|N_\epsilon^S(O)| = max\{minPoints, F * \text{ expDen}(d)\}$$

con d sottospazio con densità media minore.
Definizione: monotonicità della densità
Per ogni coppia di spazi S e T tali che $S \subseteq T$ e ogni oggetto O il numero di oggetti nel ϵ-vicinato di O in T è limitato dal numero di oggetti nel ϵ-vicinato di O in S:

$$|N_\epsilon^S(O)| \geq |N_\epsilon^T(O)|$$

Mostriamo un esempio di in-process removal: supponiamo di avere un cluster nel sottospazio {1,2,5}, i sottospazi colorati in azzurro sono quelli che vengono successivamente ristretti per arrivare a trovare il cluster. Rispetto a un approccio tradizionale di tipo breadth-first non è necessario effettuare il clustering sui sottospazi {1}, {2}, {5}, {1,2}, {1,5} e {2,5} per arrivare a trovare il cluster in {1,2,5}. Una volta trovato questo cluster è l'unico non ridondante. Possiamo quindi potare tutte le sue proiezioni in sottospazi di piu' bassa dimensionalità (indicate con una X rossa).

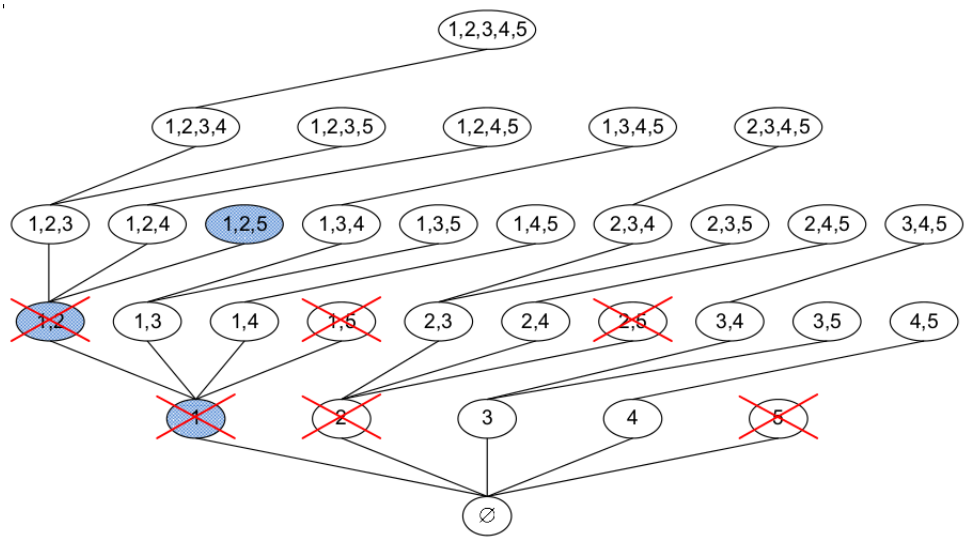
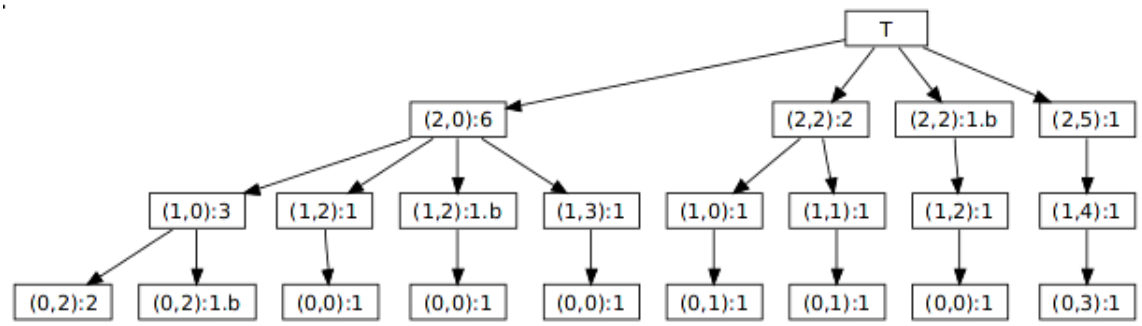


Figura 2.1: Visita depth-first con in-process removal delle ridondanze ([4])

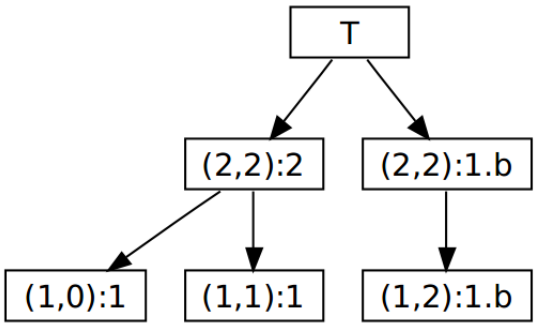
2.3.1 L'algoritmo

Il mining dello spazio avviene per gran parte del tempo lavorando esclusivamente sullo SCY-tree. I dati vengono acceduti esclusivamente durante la fase finale di clustering vero e proprio ovvero quando viene verificata la definizione di cluster sui sottospazi candidati (che sono quelli che non sono stati tagliati nelle fasi precedenti).

- **FASE 1: restrizione SCY-tree**
  Come già accennato l'algoritmo procede visitando lo spazio (ovvero lo SCY-tree) con una strategia Depth-First. La prima fase è quella di restrizione dello SCY-tree sul descrittore (d, i). Restringere lo SCY-tree significa creare un nuovo SCY-tree che è ottenuto rimuovendo dallo SCY-tree originale il livello corrispondente alla dimensione d (lo SCY-tree è ridotto di un livello) e i rami che non contengono al loro interno un nodo (d, i).
  Questa operazione tradotta sul dataset significa restringerlo ai soli punti che nella dimensione d si trovano nell'intervallo i.
  Esempio:



(a) iniziale



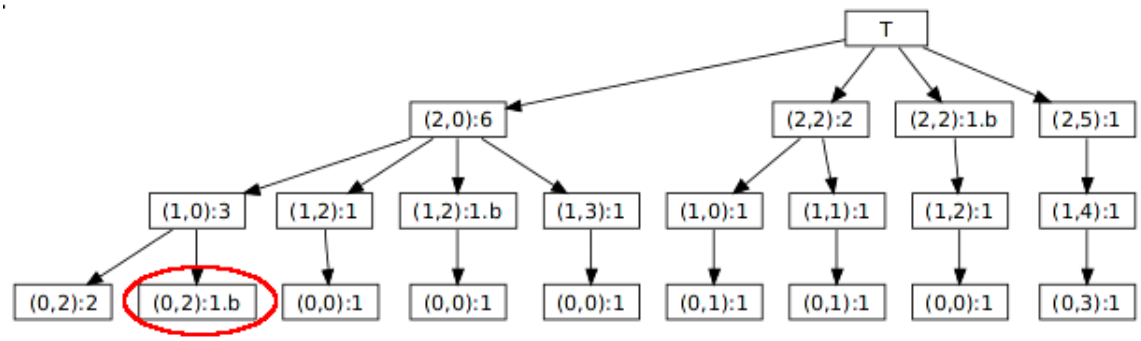
(b) ristretto su (0,1)

Figura 2.2: Restrizione SCY-tree

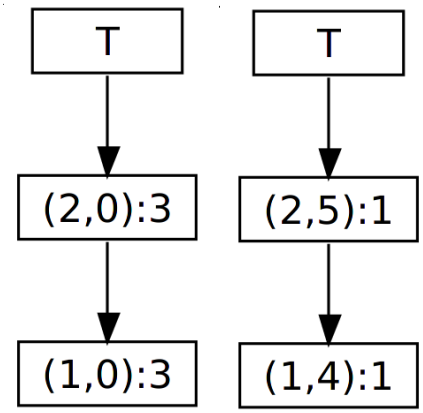
In figura 2.2 è mostrato l'albero prima e dopo la restrizione sul descrittore $(0, 1)$, come si può vedere sono presenti solo i livelli con dimensione 1 e 2 e i rami che nell'albero originario contenevano il descrittore $(0, 1)$

- **FASE 2: unione SCY-tree**
  Una delle proprietà dei cluster è la massimalità. In questa fase si controlla se sia possibile espandere la regione di spazio in esame unendola a regioni vicine. Ciò significa verificare se lo SCY-tree possa essere unito ad altri SCY-tree: due SCY-tree A e B possono essere uniti se esistono oggetti nella regione al bordo che si trova tra le due regioni A e B. Questa proprietà è facilmente verificabile ricercando nodi s-connettori nello SCY-tree iniziale (quello da cui è stato generato lo SCY-tree ristretto).
  Unire due SCY-tree significa inserire tutti i rami di uno dentro l'altro, se i nodi non esistono vengono aggiunti, se esistono se ne sommano i contatori.

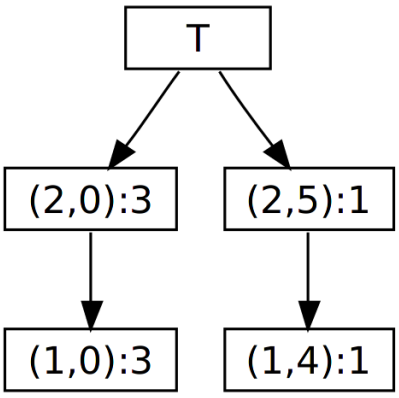
Esempio:



(a) SCY-tree originario



(b) SCY-tree $(0,2)$ (c) SCY-tree $(0,3)$



(d) merge SCY-tree

Figura 2.3: Merge SCY-tree

In figura 2.3 viene mostrato lo SCY-tree originario (a), gli SCY-tree ristretti in $(0, 2)$ e $(0, 3)$ (b,c) e la loro unione che è possibile effettuare perchè esiste un nodo s-connector nell'albero originario (cerchiato di rosso in (a)).

- **FASE 3: pruning dell'albero**
  In questa fase viene confrontato il numero degli oggetti nella regione di spazio rappresentata dallo SCY-tree (parametro associato al nodo radice) con un parametro di input dell'algoritmo, *minSize*, la soglia minima di oggetti che un sottospazio deve avere per potersi "candidare" a cluster; se il numero degli oggetti è minore della soglia lo SCY-tree viene tagliato. In questo modo la ricorsione si arresta e si passa al nodo successivo dell'albero di origine.
  Si noti che se il numero di oggetti di uno SCY-tree A è inferiore alla soglia allora anche il numero degli oggetti di qualunque SCY-tree che sia generato da una restrizione su un qualunque descrittore di A sarà sicuramente minore della soglia.

- **FASE 4: in-process removal**
  Una volta arrivati in una situazione dove, o ci troviamo nello spazio dimensionalmente più grande oppure l'albero è stato potato (fase 3) si passa alla fase successiva dell'algoritmo, si torna indietro nei sottospazi per verificare se possono nascondere cluster.
  Prima di effettuare il clustering vero e proprio si effettua l'in-process removal: si va a verificare se nel insieme dei cluster individuati ne esistono di appartenenti a una sovradimensione di quella che stiamo analizzando, che coprono la regione in cui ci troviamo. In tal caso lo SCY-tree attuale viene tagliato, avendo cosi evitato una successiva costosa scansione di densità che avrebbe trovato un cluster ridondante.
  La fase successiva è la vera e propria fase in cui si vanno a verificare le condizioni che un cluster deve rispettare

- **FASE 5: clustering**
  In questa fase si effettua il clustering vero e proprio ed è solo in questa fase che è necessario accedere al dataset per recuperare gli oggetti (finora si è lavorato esclusivamente sullo SCY-tree).
  Si va ad applicare un algoritmo di subspace cluster density-based (es. *DBSCAN*).

Si noti che nonostante l'algoritmo di density scan sia un algoritmo costoso (per via delle numerose computazioni del vicinato) nelle fasi precedenti si è andato a tagliare quelle regioni che nascondevano cluster ridondanti o che non contenevano un numero sufficiente di punti, quindi la scansione viene eseguita solo su "poche" regioni dello spazio permettendo all'algoritmo di rimanere nel complesso efficiente.

I risultati sperimentali [2] dimostrano come INSCY presenti un ottimo compromesso tra precisione nella rilevazione dei cluster e tempi di esecuzione. Alla luce di questo si è scelto di usarlo come base per lo sviluppo dell'algoritmo **dINSCY** che ne rappresenta una versione distribuita.
Nel prossimo capitolo descriveremo in dettaglio l'algoritmo sviluppato, dapprima nella sua versione sequenziale scritta nel linguaggio C++ e poi nella versione distribuita utilizzando apposite librerie MPI (che verranno brevemente descritte in Appendice A.

Viene ora mostrato un esempio di esecuzione dell'algoritmo INSCY su un piccolo dataset di prova.

2.4 INSCY: esempio di esecuzione

Di seguito verra' mostrato un tracciato di esecuzione dell'algoritmo, che verra' suddiviso in due fasi:

- **Creazione SCY-tree**
  In questa fase si crea la struttura dati principale utilizzata dall'algoritmo per il mining dello spazio multidimensionale: lo SCY-tree. I dati sugli oggetti presenti nello spazio vengono prelevati da un file di input testuale.
- **Esecuzione dell'algoritmo sullo SCY-tree creato**
  Questa fase rappresenta la parte di esecuzione vera e propria dell'algoritmo.

I parametri di input per l'algoritmo sono:

- gridsize=10: la dimensione di ogni cella della griglia
- eps=2.00: la dimensione del vicinato di un punto (utilizzata per definire i bordi delle celle e il vicinato durante dbscan)

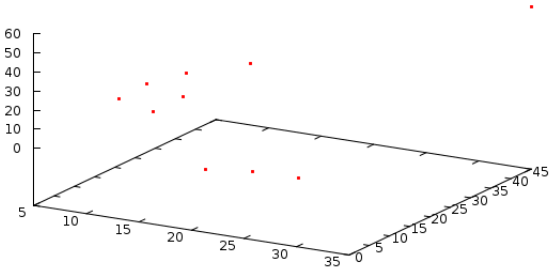
- minpts=2: il numero minimo di elementi che un cluster deve avere
- minsize=2: il numero minimo di oggetti che devono essere presenti in un sottospazio
- density=1: la minima densità interna di un cluster

Prima di passare alla spiegazione vera e propria andrò a introdurre il dataset utilizzato in questo esempio.

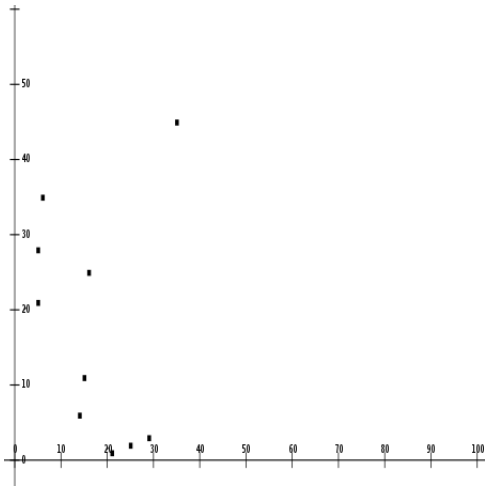
2.4.1 Il dataset

Il dataset utilizzato per questo esempio di esecuzione e' stato creato ad-hoc, esso è composto da 10 oggetti che si trovano in uno spazio tridimensionale. Di seguito viene mostrato il dataset e viene presentata la distribuzione degli oggetti nello spazio attraverso una serie di scatter plot.

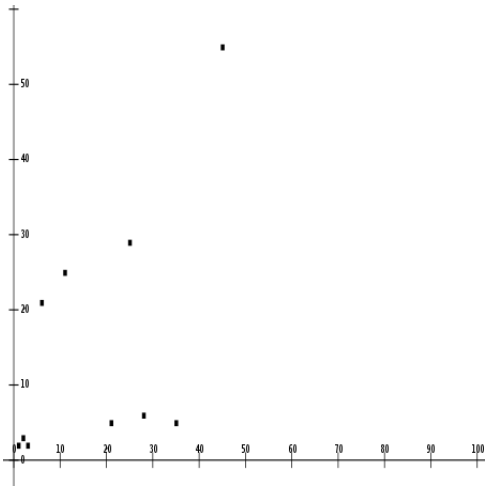
X Y Z
{14,06,21}
{21,01,02}
{15,11,25}
{25,02,03}
{05,21,05}
{16,25,29}
{25,45,55}
{29,03,02}
{05,28,06}
{06,35,05}



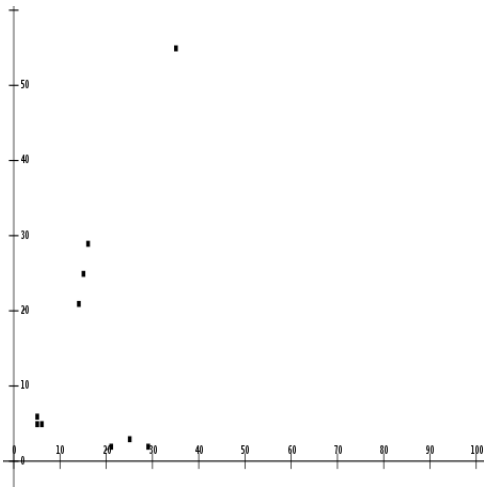
(a) 3d scatter-plot



(b) 2d scatter-plot x-y



(c) 2d scatter-plot y-¡



(d) 2d scatter-plot x-z

Figura 2.4: Dataset plots

2.4.2 Creazione SCY-tree

La creazione dello SCY-tree viene effettuata leggendo il file di input (formattato come da 4.3) e andando a inserire (o aggiornare), per ogni oggetto, tanti SCY-node quante sono le dimensioni di ogni oggeto. Uno SCY-node è una tripla (d, i, c) dove d rappresenta la dimensione, i rappresenta l'intervallo per quella dimensione e c identifica il numero degli oggetti che nella dimensione d si trovano nell'intervallo i. Andrò a mostrare come si modifica lo SCY-tree all'inserimento di ogni dimensione per i primi due oggetti del dataset, poi mostrero' l'inserimento di ulteriori due oggetti senza mostrarne l'inserimento di ogni dimensione, mostrando infine allo SCY-tree finale creato dall'algoritmo.

Oggetto {14,06,21}

Andiamo ora ad inserire il primo oggetto del dataset dimensione per dimensione. Lo SCY-tree, dopo l'inserimento è composto dalla radice e da un unico nodo che rappresenta la dimensione inserita e l'intervallo in cui il valore va a inserirsi (che viene determinata considerando il parametro di input *gridsize*.

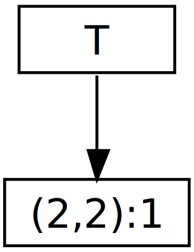


Figura 2.5: Inserimento oggetto 0, dimensione 2 nello SCY-tree

Il nodo che si riferisce alla dimensione 1 dell'oggetto viene inserito come figlio dell'ultimo nodo inserito.

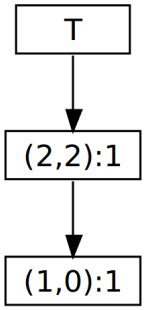


Figura 2.6: Inserimento oggetto 0, dimensione 1 nello SCY-tree

Come per la dimensione 1 anche il nodo relativo alla dimensione 0 viene inserito come figlio dell'ultimo nodo inserito. L'albero cosi composto rappresenta quindi l'oggetto inserito e questo è corretto in quanto in uno SCY-tree ogni oggetto è univocamente rappresentato da un cammino che va da una foglia fino alla radice.

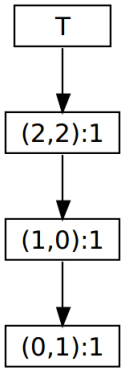


Figura 2.7: Inserimento oggetto 0, dimensione 0 nello SCY-tree

Oggetto {21,01,02}

Vediamo ora l'inserimento del secondo oggetto del dataset.
Per la dimensione 2 viene inserito uno SCY-node all'albero come secondo figlio della radice.

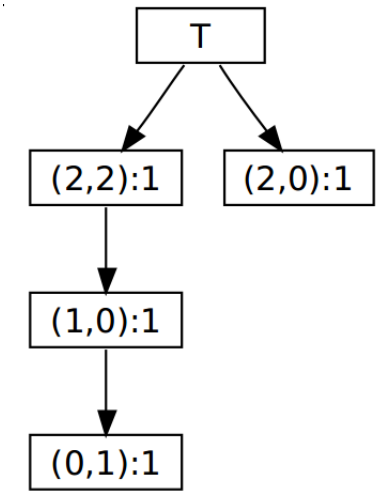


Figura 2.8: Inserimento oggetto 1, dimensione 2 nello SCY-tree

Il nodo relativo alla dimensione 1 viene inserito come figlio del nodo precedente.

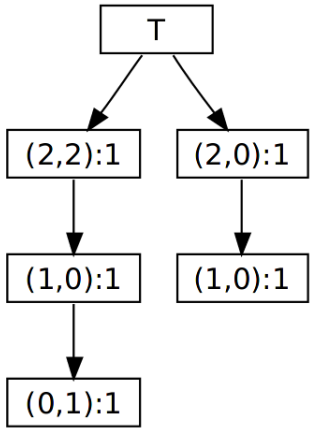


Figura 2.9: Inserimento oggetto 1, dimensione 1 nello SCY-tree

Anche il nodo relativo alla dimensione 0 viene inserito come figlio del nodo relativo alla dimensione 1.

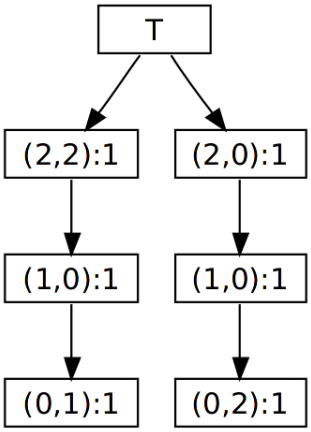


Figura 2.10: Inserimento oggetto 1, dimensione 0 nello SCY-tree

Oggetto {15,11,25}

Come possiamo notare dall'inserimento di questo oggetto, nella dimensione 2 l'oggetto si inserisce in un intervallo dove vi è già un oggetto, il contatore del relativo SCY-node viene allora incrementato di uno, per quanto riguarda le dimensioni 1 e 0 viene creato un altro ramo dell'albero che contiene gli SCY-node relativi.

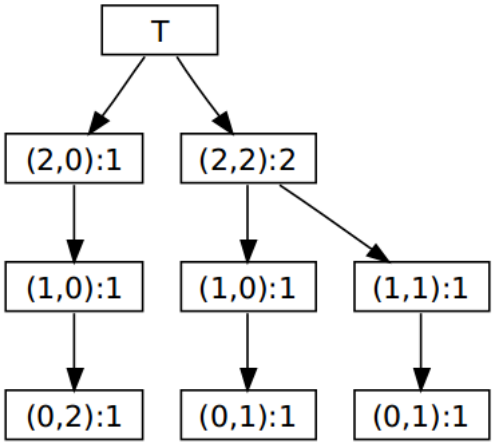


Figura 2.11: Inserimento oggetto 2 nello SCY-tree

Oggetto {25,02,03}

In questo caso l'oggetto va a inserirsi per tutte e tre le dimensioni in intervalli dove è già presente un altro oggetto, viene allora aggiornato il ramo dell'albero relativo incrementando i contatori di tutti gli SCY-node di uno. Per la nostra struttura indice questo oggetto non viene distinto dall'oggetto {21,01,02} in quanto risiedono, per ogni dimensione, negli stessi intervalli..

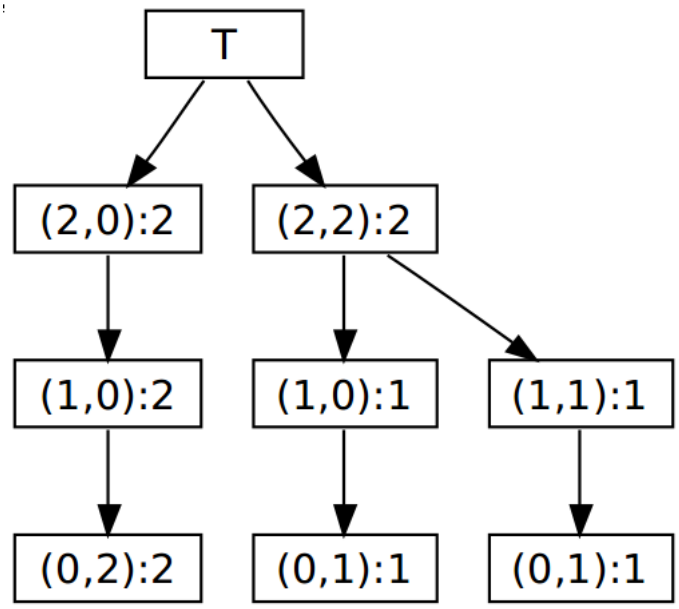


Figura 2.12: Inserimento oggetto 3 nello SCY-tree

SCY-tree finale creato dall'algoritmo

Di seguito viene mostrato lo SCY-tree ottenuto dopo l'inserimento dei rimanenti punti. Notiamo la presenza di alcuni SCY-node contrassegnati da *.b*, questi sono speciali nodi chiamati s-connettori che rappresentano punti collocati sul bordo tra due celle e hanno la funzione di permettere il merge di celle adiacenti, necessario per rispettare la definizione di subspace cluster. Per definire se un punto è al bordo o meno si utilizza il parametro di input *eps*.

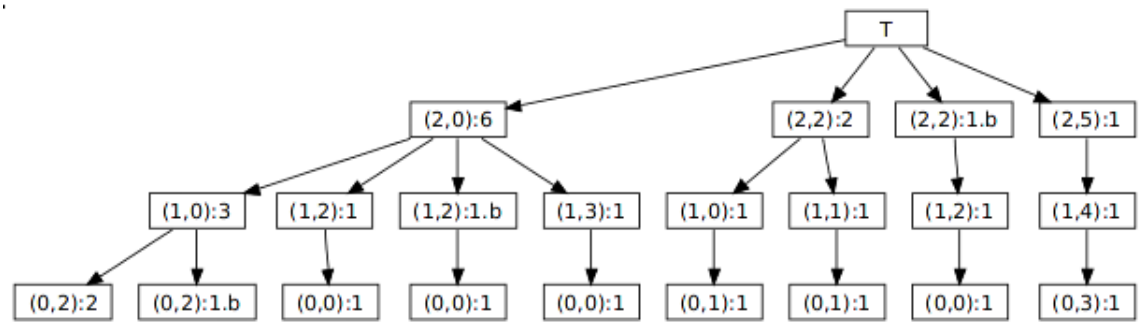


Figura 2.13: SCY-tree finale

2.4.3 L'algoritmo

Di seguito viene presentato il tracciato di esecuzione dell'algoritmo attraverso una serie di chiamate di pseudofunzioni. I nodi vengono rappresentati attraverso una coppia (d, i) con d dimensione e i intervallo. Il nodo rappresentante l'unione di 2 o più intervalli nella dimensione d viene indicato con $(d, i_1 - i_2)$ Indichiamo con *node1; . . . ; noden* lo SCY-tree ottenuto partendo dallo SCY-tree iniziale e restringendo sequenzialmente sui nodi node1,. . . , noden. Le funzioni utilizzate sono:

- *restrict(node1; . . . ; noden)*: questa funzione restringe lo SCY-tree *node1; . . . ; noden-1* rispetto al nodo *noden*. Restringere rispetto al nodo (d,i) significa:
  - tagliare i rami dell'albero che non contengono nodi (d,i).
  - eliminare il livello dell'albero che corrisponde alla dimensione d
- *findMerges(node1; . . . ; noden)*: questa funzione ricerca SCY-tree che possano essere uniti con lo SCY-tree *node1; . . . ; noden* e effettua l'unione. Due

alberi possono essere uniti se nell'albero da cui sono stati generati (mediante restrizione) esiste un nodo s-connettore tra di loro, questo significa che esiste almeno un oggetto che si trova al bordo tra i due intervalli e nella dimensione che i due alberi rappresentano.

- *checkPruning(node1; ...; noden)*: questa funzione verifica il numero di oggetti che lo SCY-tree *node1; ...; noden* referenzia e se questo è sotto una certa soglia taglia lo SCY-tree.

- *checkRedundancy(node1; ...; noden)*: questa funzione verifica se lo SCY-tree *node1; ...; noden* è ridondante rispetto a qualche cluster già individuato. Se si taglia lo SCY-tree.

- *mine(node1; ...; noden)*: questa funzione effetua il mining vero e proprio sul dataset rappresentato dallo SCY-tree *node1; ...; noden* allo scopo di trovare eventuali cluster. Per questa fase viene utilizzando l'algoritmo DBSCAN.

Per aiutare a comprendere meglio i passi dell'algoritmo andiamo a vedere l'albero di visita (restrizione) dello spazio. Nell'albero vengono indicati:

- **regioni dello spazio in cui vengono trovati cluster**: nodi in rosso (con indicazione del cluster individuato)

- **regioni dello spazio che risultano coperte da cluster già individuati**: nodi in blu (con riferimento al cluster)

- **rami dell'albero potati dall'algoritmo (funzione *checkPruning()*)**: X grigie

- **merge tra spazi effettuati dall'algoritmo**: linea che collega due nodi

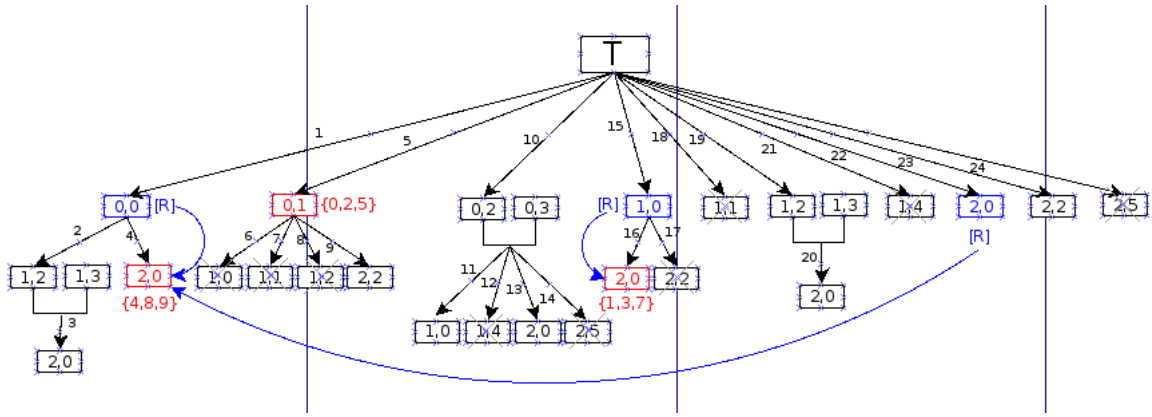


Figura 2.14: Albero di visita dello spazio

Con i numeri sulle frecce viene indicato l'ordine di restrizione dello spazio, in pratica visitando in profondità l'albero si restringe lo spazio verificando nel contempo

possibilità di merge tra regioni dello spazio

possibilità di potatura dell'albero (stop della ricorsione) se il numero di elementi nella regione dello spazio è minore di una certa soglia

risalendo l'albero invece si verifica che la regione non sia coperta da cluster già individuati, se questa verifica da esito positivo allora la zona è candidata al mining vero e proprio, viene quindi applicato l'algoritmo dbscan per la ricerca di eventuali cluster.

Passiamo ora a mostrare in maniera dettagliata i passi compiuti dall'algoritmo, mostrando come si modifica lo SCY-tree durante l'esecuzione.

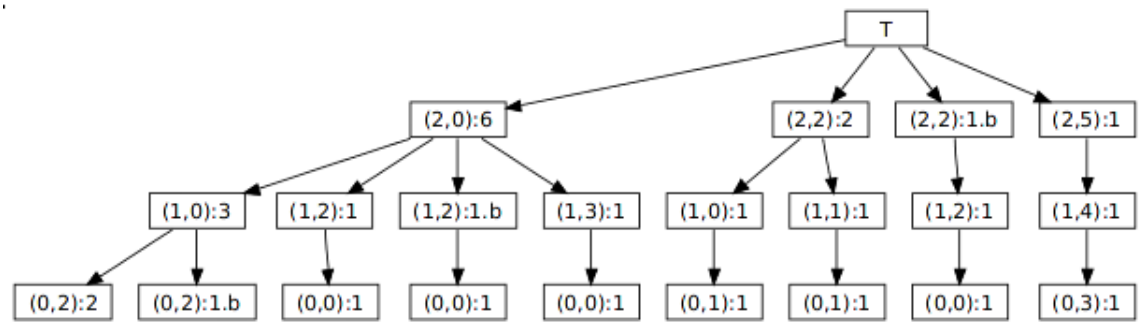


Figura 2.15: SCY-tree iniziale

restrict(0,0)

In questo primo passo si effettua la restrizione sul primo nodo, in ordine lessicografico, dello SCY-tree iniziale.

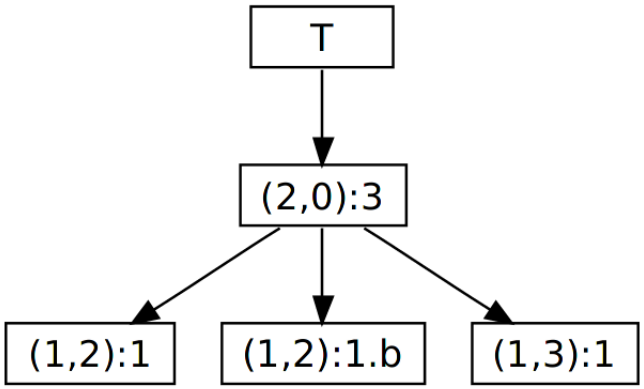


Figura 2.16: Esecuzione restrict(0,0)

findMerges(0,0) = {}

L'algoritmo cerca possibili merge dell'albero, in questo caso non esiste nessun s-connettore nell'albero di origine quindi non è possibile nessuna unione.

checkPruning(0,0)=false

L'algoritmo verifica se l'albero possa essere potato o meno, il numero di oggetti referenziati è 3 che è maggiore della soglia *minsize* quindi l'albero non viene potato.

restrict(0,0;1,2)

L'algoritmo procede in profondità andando a restringere sul primo nodo in ordine lessicografico dell'albero.

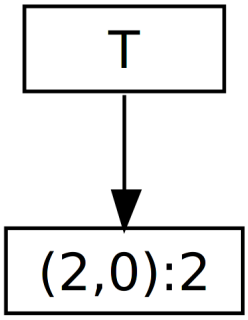
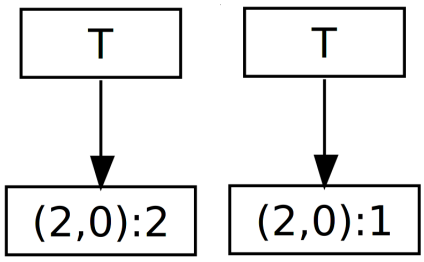


Figura 2.17: Esecuzione restrict(0,0;1,2)

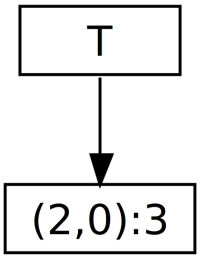
findMerges(0,0;1,2) = {0,0;1,2 ↔ 0,0;1,3}

Questa volta l'albero può essere unito all'albero *0,0-1,3* in quanto nell'albero superiore *0,0* esiste il nodo s-connettore *(1,2)* che collega i due nodi da cui questi alberi provegono.



(a) primo SCY-tree

(b) secondo SCY-tree



(c) unione SCY-tree

Figura 2.18: Esecuzione merge 0,0;1,2 ↔ 0,0;1,3

checkPruning(0,0;1,2-3)=false

Anche qui l'albero referenzia un numero sufficiente di oggetti e quindi non viene tagliato.

restrict(0,0;1,2-3;2,0)

L'algoritmo procede nella restrizione in profondità.

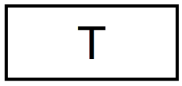


Figura 2.19: Esecuzione restrict(0,0;1,2-3;2,0)

checkPruning(0,0;1,2-3;2,0)=false

checkRedundancy(0,0;1,2-3;2,0)=false

Non sono più possibili restrizioni in profondità, si passa quindi alla fase successiva dell'algoritmo in cui si va a verificare se il sottospazio ristretto sia già coperto o meno da un cluster già individuato. In questo caso, non avendo ancora individuato cluster ovviamente la risposta è negativa.

mine(0,0;1,2-3;2,0)={}

Il passo successivo, se l'albero non viene potato dal check delle ridondanze e il mining vero e proprio che consiste nell'accedere al dataset, recuperare gli oggetti referenziati dallo SCY-tree (per far questo si usa in un certo senso un algoritmo inverso a quello di creazione dello SCY-tree) e applicare sugli oggetti dbscan. In questo caso non vengono trovati cluster da dbscan.

checkRedundancy(0,0;1,2-3)=false

Si comincia ora a risalire l'albero ripetendo il check della ridondanza, anche in questo caso l'albero non viene potato.

mine(0,0;1,2-3)={}

Il mine non restituisce nessun cluster.

restrict(0,0;2,0)

Si passa a restringere sul nodo successivo dell'albero *0,0* e si procede in profondità.

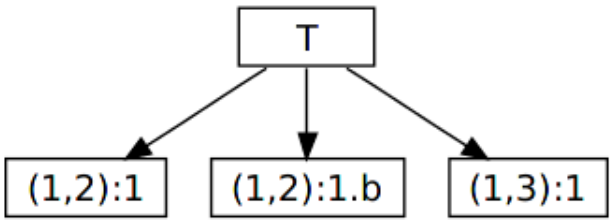


Figura 2.20: Esecuzione restrict(0,0;2,0)

checkPruning(0,0;2,0)=false

Nella fase di check del pruning l'albero non viene potato.

checkRedundancy(0,0;2,0)=false

Nella fase di check della ridondanza l'albero non viene potato.

mine(0,0;2,0)={p(4),p(8),p(9)}

Dbscan trova in questa regione dello spazio un cluster formato dai punti 4,8 e 9

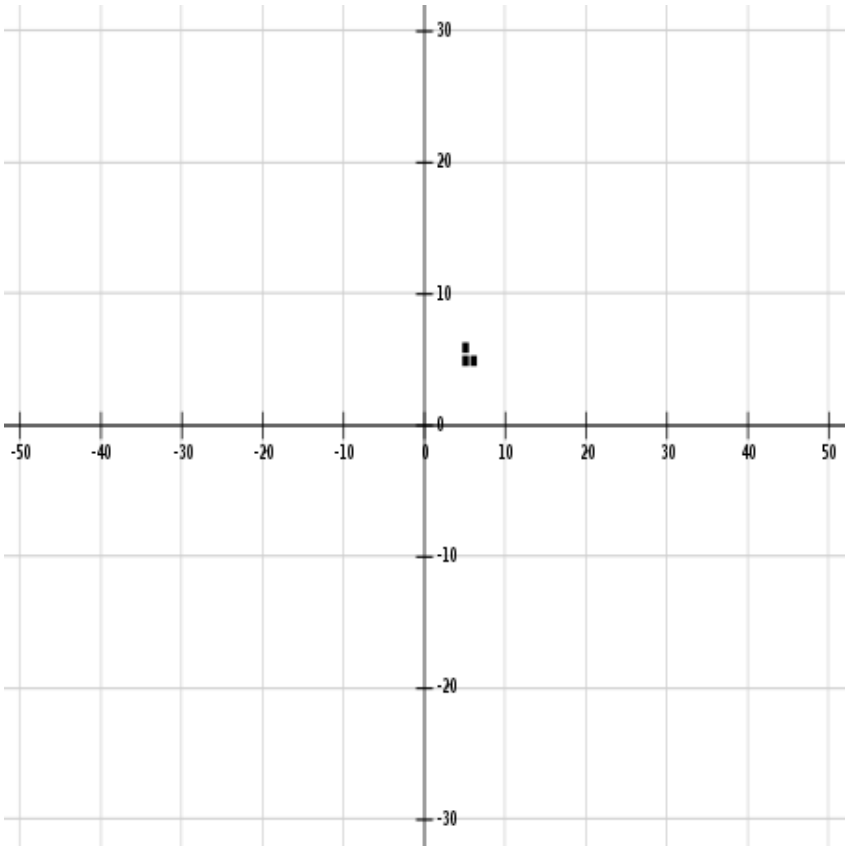


Figura 2.21: Cluster individuato in 0,0;2,0

checkRedundancy(0,0)=true

Risalendo la sequenza delle restrizioni l'albero ristretto in 0,0 viene potato in quanto già coperto dal cluster appena trovato.

restrict(0,1)

Si passa a restringere sul nodo successivo dell'albero iniziale.

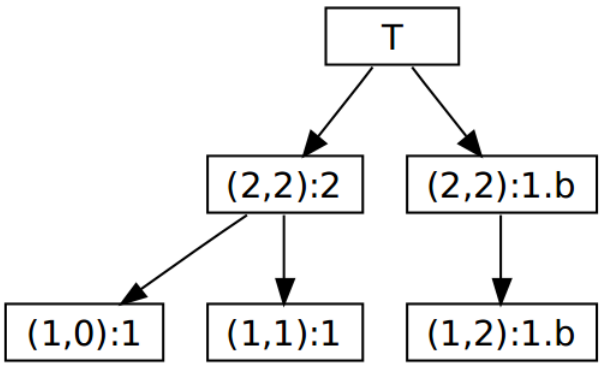


Figura 2.22: Esecuzione restrict(0,1)

findMerges(0,1) = {}

L'albero ristretto non può essere unito ad altri alberi.

checkPruning(0,1)=false

L'albero ristretto non viene potato in quanto referenzia 3 oggetti.

restrict(0,1;1,0)

Si procede a restringere in profondità.

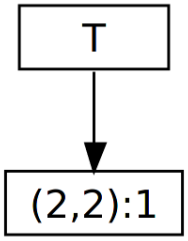


Figura 2.23: Esecuzione restrict(0,1;1,0)

findMerges(0,1;1,0) = {}

Nessuna unione è possibile.

checkPruning(0,1;1,0)=true

L'albero viene potato in quanto referenzia 1 oggetto solo.

restrict(0,1;1,1)

L'albero è stato potato quindi si risale e si restringe sul nodo successivo dell'albero 0,1

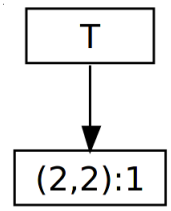


Figura 2.24: Esecuzione restrict(0,1;1,1)

findMerges(0,1;1,1) = {}

Nessuna unione è possibile.

checkPruning(0,1;1,1)=true

Anche in questo caso l'albero referenzia un solo oggetto quindi viene potato.

restrict(0,1;1,2)

Si procede con il nodo successivo dell'albero 0,1

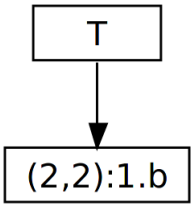


Figura 2.25: Esecuzione restrict(0,1;1,2)

findMerges(0,1;1,2) = {}

checkPruning(0,1;1,2)=true

L'albero viene potato.

restrict(0,1;2,2)

Si procede restringendo sull'ultimo nodo dell'albero 0,1.

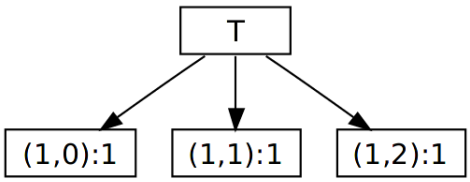


Figura 2.26: Esecuzione restrict(0,1;2,2)

findMerges(0,1;2,2) = {}

Anche per questo albero nessuna unione è possibile.

checkPruning(0,1;2,2)=false

L'albero referenzia 3 punti quindi non viene potato.

checkRedundancy(0,1;2,2)=false

Non esistono cluster che già coprano questo albero.

mine(0,0;2,2)={}

Dbscan non trova nessun cluster.

checkRedundancy(0,1)=false

Risalendo l'albero si verifica la ridondanza rispetto ai cluster trovati e questa da esito negativo.

mine(0,1)={p(0),p(2),p(5)}

Dbscan in questa regione di spazio trova un secondo cluster composto dai punti 0,2 e 5

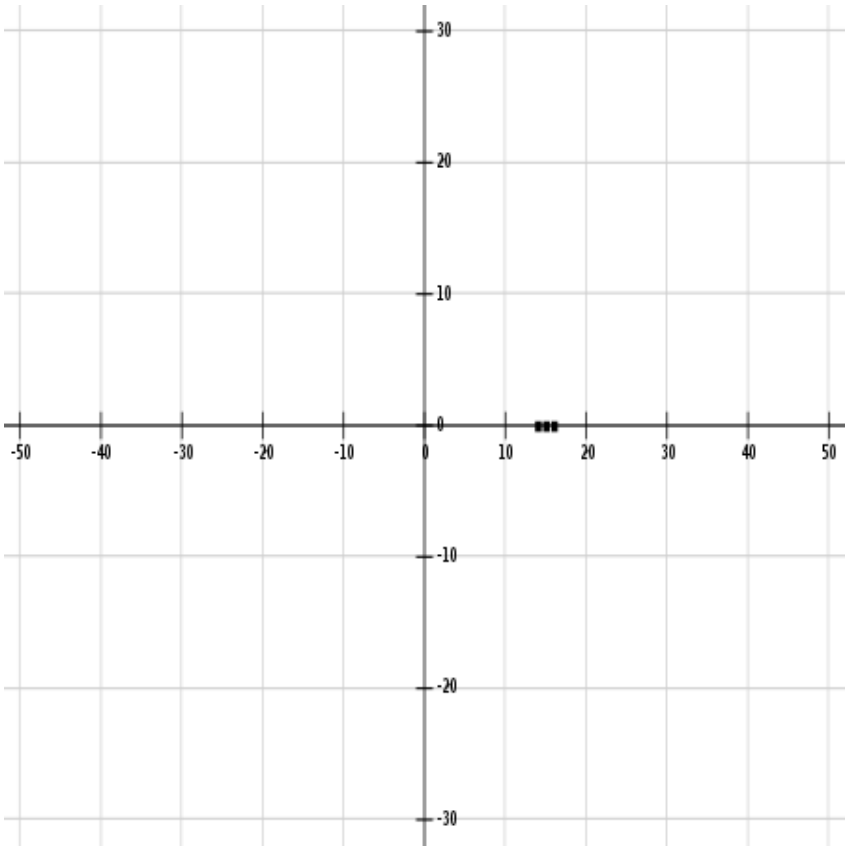


Figura 2.27: Cluster individuato in 0,1

restrict(0,2)

Procede la restrizione sull'albero iniziale.

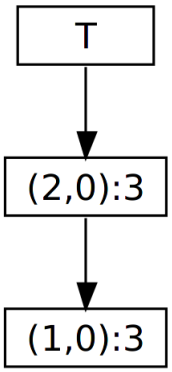
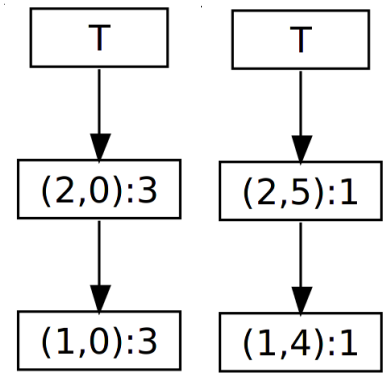


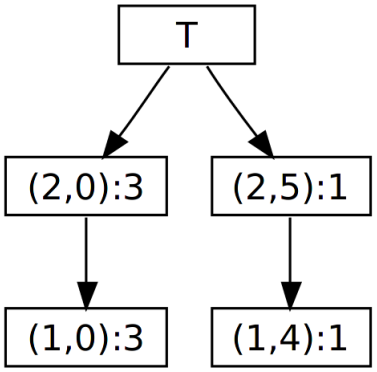
Figura 2.28: Esecuzione restrict(0,2)

findMerges(0,2) = {0,2 ↔ 0,3}

L'albero 0,2 può essere unito all'albero 0,3 in quanto nell'albero iniziale esiste il l's-connettore *0,2*



(a) primo SCY-tree (b) secondo SCY-tree



(c) unione SCY-tree

Figura 2.29: Esecuzione merge 0,2 ↔ 0,3

checkPruning(0,2-3)=false

L'albero referenzia 4 punti quindi non viene potato.

restrict(0,2-3;1,0)

Procede la restrizione in profondità.

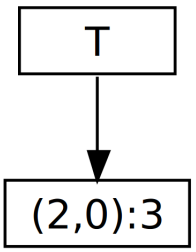


Figura 2.30: Esecuzione restrict(0,2-3;1,0)

findMerges(0,2-3;1,0) = {}

Questo albero non può essere unito con altri alberi.

checkPruning(0,2-3;1,0)=false

L'albero referenzia 3 oggetti quindi non viene potato.

restrict(0,2-3;1,0;2,0)

Continua la restrizione in profondità.

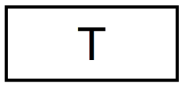


Figura 2.31: Esecuzione restrict(0,2-3;1,0;2,0)

findMerges(0,2-3;1,0;2,0) = {}

Non sono possibili unioni.

checkPruning(0,2-3;1,0;2,0)=false

I punti referenziati dall'albero sopra risiedono tutti in questo intervallo/dimensione quindi l'albero non viene potato.

checkRedundancy(0,2-3;1,0;2,0)=false

Non esistono cluster che coprano questa regione di spazio.

mine(0,2-3;1,0;2,0)={}

Gli oggetti referenziati da questo albero non rappresentano un cluster.

checkRedundancy(0,2-3;1,0)=false

Non ci sono cluster che coprono questa regione di spazio.

mine(0,2-3;1,0)={}

Dbscan non trova nessun cluster.

restrict(0,2-3;1,4)

Continua la restrizione, da ora in poi verranno commentati solo i punti salienti dell'algoritmo.

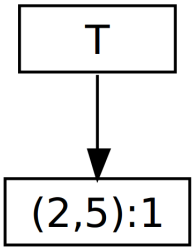


Figura 2.32: Esecuzione restrict(0,2-3;1,4)

findMerges(0,2-3;1,4) = {}

checkPruning(0,2-3;1,4)=true

Questo albero referenzia un unico punto, viene quindi potato.

restrict(0,2-3;2,0)

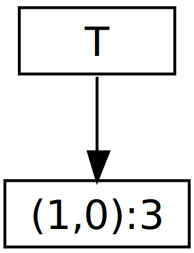


Figura 2.33: Esecuzione restrict(0,2-3;2,0)

findMerges(0,2-3;2,0) = {}

checkPruning(0,2-3;2,0)=false

checkRedundancy(0,2-3;2,0)=false

mine(0,2-3;2,0)={}

restrict(0,2-3;2,5)

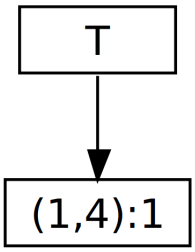


Figura 2.34: Esecuzione restrict(0,2-3;2,5)

findMerges(0,2-3;2,5) = {}

checkPruning(0,2-3;2,5)=true

Questo albero referenzia solo un punto quindi viene potato.

checkRedundancy(0,2-3)=false

mine(0,2-3)={}

restrict(1,0)

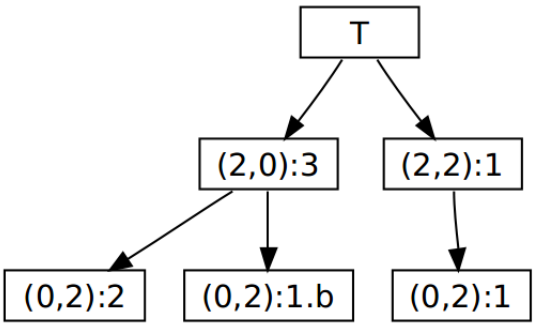


Figura 2.35: Esecuzione restrict(1,0)

findMerges(1,0) = {}

checkPruning(1,0)=false

restrict(1,0;2,0)

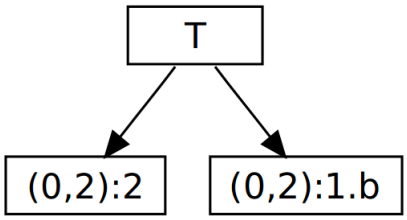


Figura 2.36: Esecuzione restrict(1,0;2,0)

findMerges(1,0;2,0) = {}

checkPruning(1,0;2,0)=false

checkRedundancy(1,0;2,0)=false

mine(1,0;2,0)={p(1),p(3),p(7)}

Dbscan trova in questa regione di spazio l'ultimo cluster composto dagli oggetti 1,3 e 7.

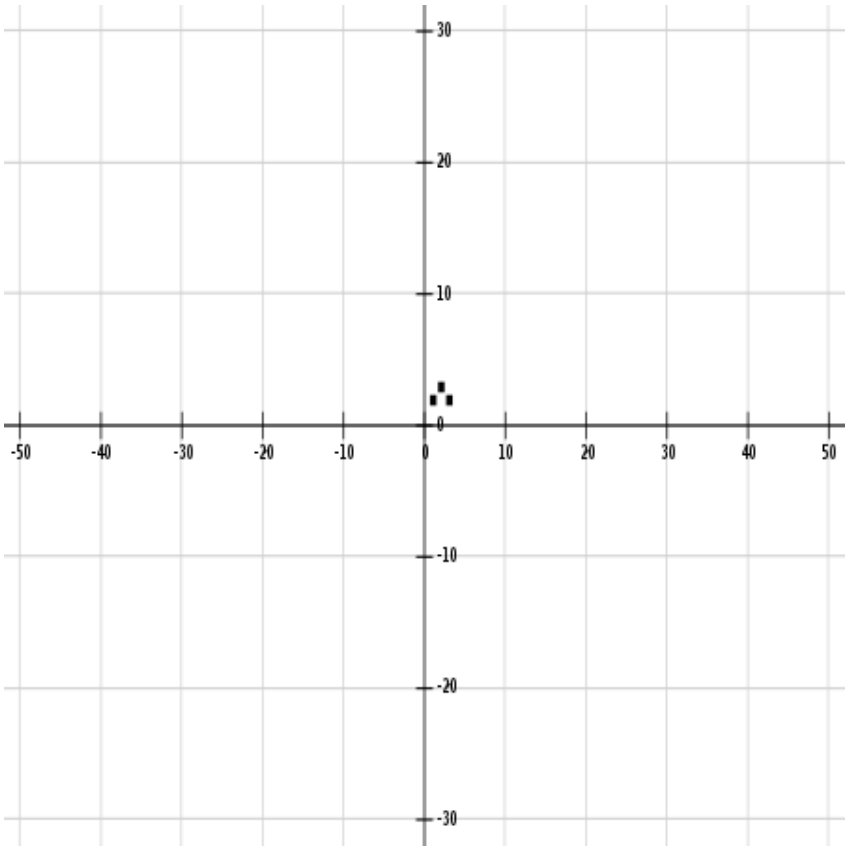


Figura 2.37: Cluster individuato in 1,0;2,0

restrict(1,0;2,2)

Prosegue la sequenza di restrizioni.

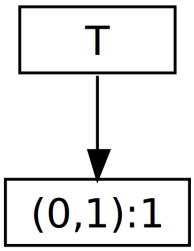


Figura 2.38: Esecuzione restrict(1,0;2,2)

findMerges(1,0;2,2) = {}

checkPruning(1,0;2,2)=true

L'albero referenzia un unico punto, viene quindi potato dall'algoritmo.

checkRedundancy(1,0)=true

Si risale e si va a verificare se esistono cluster che coprono questa regione di spazio, la risposta è positiva, l'ultimo cluster trovato copre questo albero.

restrict(1,1)

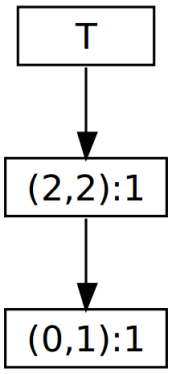


Figura 2.39: Esecuzione restrict(1,1)

findMerges(1,1) = {}

checkPruning(1,1)=true

Questo albero referenzia un unico punto, viene quindi potato.

restrict(1,2)

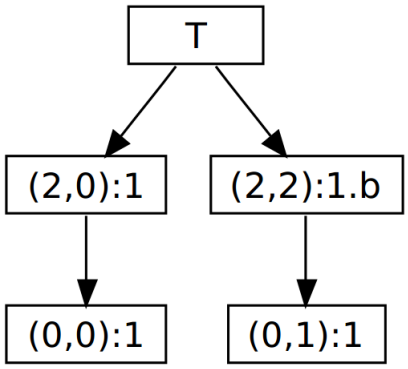
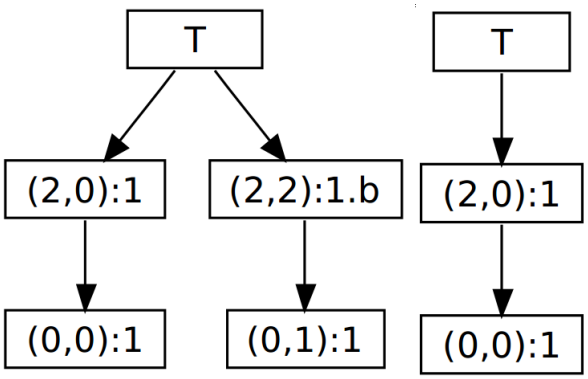


Figura 2.40: Esecuzione restrict(1,2)

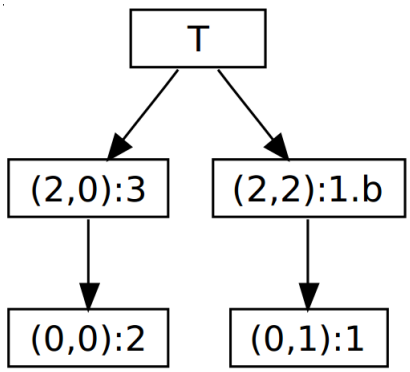
findMerges(1,2) = {1,2 ↔ 1,3}

Per questo albero esiste un possibile merge: l'albero 1,3, se infatti facciamo un passo indietro vediamo che nell'albero iniziale esiste un nodo s-connettore 1,2.



(a) primo SCY-tree

(b) secondo SCY-tree



(c) unione SCY-tree

Figura 2.41: Esecuzione merge 1,2 $\leftrightarrow$ 1,3

checkPruning(1,2-3)=false

restrict(1,2-3;2,0)

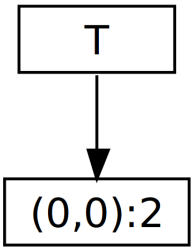


Figura 2.42: Esecuzione restrict(1,2-3;2,0)

findMerges(1,2-3;2,0) = {}

checkPruning(1,2-3;2,0)=false

checkRedundancy(1,2-3;2,0)=false

mine(1,2-3;2,0)={}

checkRedundancy(1,2-3)=false

mine(1,2-3)={}

restrict(1,4)

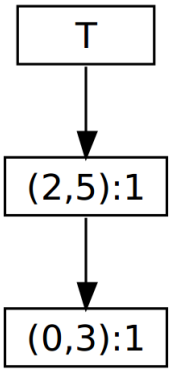


Figura 2.43: Esecuzione restrict(1,4)

findMerges(1,4) = {}

checkPruning(1,4)=false

restrict(2,0)

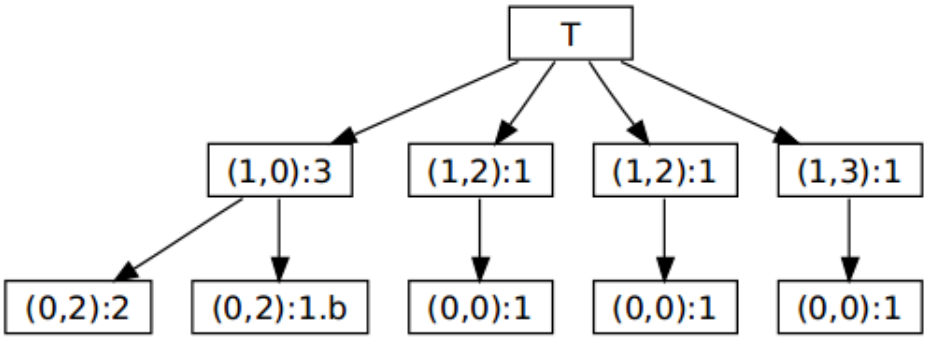


Figura 2.44: Esecuzione restrict(2,0)

findMerges(2,0) = {}

checkPruning(2,0)=false

checkRedundancy(2,0)=true

Questa regione di spazio è già coperta dall'ultimo cluster trovato.

restrict(2,2)

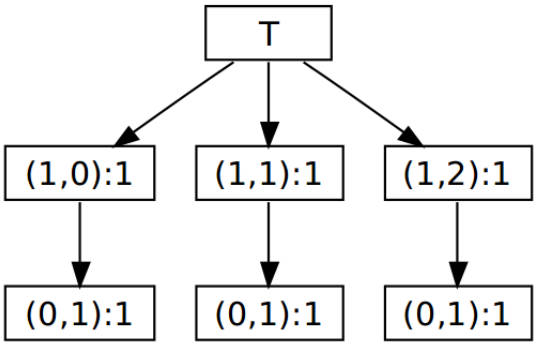


Figura 2.45: Esecuzione restrict(2,2)

findMerges(2,2) = {}

checkPruning(2,2)=false

checkRedundancy(2,2)=false

mine(2,2)={}

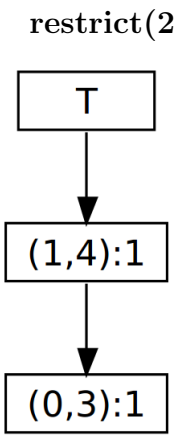


Figura 2.46: Esecuzione restrict(2,5)

findMerges(2,5) = {}

checkPruning(2,5)=true

L'albero referenzia un unico punto, viene quindi potato dall'algoritmo.

Alla fine dell'algoritmo sono stati trovati 3 cluster (sono indicati tra [] le dimensioni in cui esistono i cluster e tra {} gli indici degli oggetti contenuti nel cluster):

SC0[101] = {4,8,9}

SC1[100] = {0,2,5}

SC2[011] = {1,3,7}

Capitolo 3

Introduzione al calcolo parallelo

In questo capitolo verrà fatta una breve panoramica sul calcolo parallelo e distribuito. Si andrà dapprima a spiegare cosa è il calcolo parallelo per poi dare una panoramica della tassionomia dei sistemi di calcolo parallelo. In seguito verranno spiegati i due principali modelli di programmazione parallela concludendo con una presentazione delle principali metriche utilizzate per la valutazione delle performance degli algoritmi paralleli.

3.1 Le motivazioni

Il tradizionale modello di calcolo prevede l'esecuzione dei programmi su una macchina ad architettura di Von Neumann ovvero una macchina con una singola unità di elaborazione o CPU e una singola memoria centrale. Questo modello prende il nome di "*calcolo sequenziale*".

Il modello di calcolo parallelo e distribuito [14] si basano sull'utilizzo contemporaneo di diverse unità di elaborazione per l'esecuzione dei programmi. Sebbene oggi la distinzione tra calcolo parallelo e calcolo distribuito sia molto sottile e non esista di fatto una definizione ufficiale dei due modelli, una delle possibili definizioni vede il **calcolo parallelo** associato al modello di calcolo a memoria condivisa e il **calcolo distribuito** come il modello basato su message-passing.
Da questo punto useremo genericamete il termine "calcolo parallelo" per riferirci sia al modello di calcolo parallelo che al modello di calcolo distribuito.

Le principali motivazioni del modello di calcolo parallelo sono da ricercarsi nelle limitazioni fisiche che le architetture tradizionali odierne (basate sul modello di Van Neumann) portano: le moderne CPU sono composte da milioni di

piccolisimi componenti elettronici chiamati transistor, senza entrare nei dettagli elettronici possiamo affermare che la velocità di una CPU dipende dal numero di transistor che la compongono oltre che dalla loro tecnologia produttiva. In particolare l'aumento della velocità di clock delle CPU porta a un aumento della potenza dissipata dalle stesse e quindi del calore generato, diminuire la grandezza dei transistor permette di avere una minore quantità di calore prodotto a parità di velocità. Il limite fisico è da ricercarsi proprio nella grandezza dei transistor che non può essere diminuita all'infinito (i transistor sono formati da atomi), se non si può aumentare all'infinito la velocità delle CPU (operazione che comunque richiede anche un grande investimento economico in ricerca) perchè non far cooperare diverse CPU nell'esecuzione di un programma?

3.2 La classificazione logica

I sistemi di calcolo parallelo possono essere classificati attraverso la loro capacità di gestire flussi di istruzioni e di dati paralleli, questa classificazione prende il nome dal suo creatore: *Tassionomia di Flynn*

- **Single Instruction Single Data**: una sola unità di calcolo che esegue una sola istruzione e accetta un solo dato in ingresso ad ogni ciclo di clock, nessun parallelismo.
- **Single Instruction Multiple Data**: durante lo stesso ciclo di clock la medesima istruzione viene eseguita da più unità di calcolo su dati diversi.
- **Multiple Instruction Single Data**: diverse istruzioni vengono eseguite da diverse unità di calcolo sugli stessi dati, questa architettura è poco utilizzata in quanto non ha tante applicazioni concrete.
- **Multiple Instruction Multiple Data**: questa è l'architettura attualmente più diffusa, ogni processore può eseguire contemporaneamente agli altri istruzioni diverse su dati diversi

Una ulteriore classificazione che si può fare dei diversi sistemi di calcolo parallelo è in base all'organizzazione della memoria la quale può essere:

- **Condivisa**: la memoria è condivisa da tutti i processori quindi una modifica fatta da un processore in una porzione di memoria è immediatamente

visibile a tutti gli altri processori.
Si distinguono due modalità di accesso alla memoria: **uniforme** (UMA) in cui i tempi di accesso ad ogni locazione di memoria sono uniformi per ogni processore e **non uniforme** (NUMA) in cui ogni processore ha un accesso "preferenziale" rispetto ad altri ad alcune locazioni di memoria ovvero accede più velocemente a quelle locazioni di memoria rispetto ad altre che sono invece "preferenziali" per altri processori.

- **Distribuita**: ogni processore ha una sua porzione di memoria ad accesso esclusivo e lavora indipendentemente dagli altri processori. Ogni modifica effettuata da un processore sulla sua porzione di memoria non influenza gli altri processori e la comunicazione avviene attraverso scambio di messaggi; da qui il nome **message-passing interface**

- **Ibrida**: questa è l'architettura oggi più utilizzata: sistemi multi-processore a memoria condivisa vengono inter-connessi attraverso una rete. Si hanno tanti sistemi a memoria condivisa che comunicano tra di loro attraverso message-passing.

3.3 I sistemi di calcolo parallelo: le reti di interconnessione

Le reti di interconnessioni tra i processori che compongono un sistema parallelo possono essere classificate in **statiche** in cui ogni processore è collegato staticamente ad altri n processori (con n variabile a seconda della configurazione della rete che può essere a maglie, a stella, a ipercubo, toroidale ecc.) e **dinamiche** in cui i processori sono connessi tramite switch i quali possono riconfigurarsi per cambiare dinamicamente le connessioni.

Due importanti metriche delle reti di interconnessione sono il **fanout** e il **diametro**.
Con fanout si indica il numero di collegamenti in uscita da ogni singolo nodo della rete. Questa è una importante misura che indica la tolleranza della rete ai guasti (più il fanout di ogni nodo è grande più la rete è tollerante all'interruzione di collegamenti tra nodi). Ad esempio una rete di n nodi completamente connessa ha una grande tolleranza avendo ogni nodo ha un fanout uguale a $n - 1$.

Con diametro si indica invece la distanza minima tra i due nodi più lontani della rete. Ad esempio una rete di n nodi con topologia a bus presenta un diametro di $n-1$ essendo necessario attraversare $n-1$ archi per raggiungere dal nodo ad un estremità della rete il nodo all'altra estremità della rete.

3.4 I modelli di programmazione parallela

Vengono ora presentati alcuni modelli per la progettazione di algoritmi paralleli. A dispetto dei nomi questi modelli non fanno riferimento a specifiche architetture; possono quindi essere implementati su qualunque tipologia di calcolatore parallelo (con pro e contro specifici per ogni uno).
Definiamo con **task** le parti in cui l'algoritmo viene scomposto e che sono assegnate alle singole unità di calcolo.
Distinguiamo i modelli di programmazione parallela in base all'**interazione tra processi**:

- **Memoria condivisa**: in questo modello i diversi task condividono un'unica area di memoria alla quale accedono in maniera asincrona, non è quindi necessario esplicitare la comunicazione tra task ma controllare l'accesso alla memoria attraverso meccanismi di accesso esclusivo come *semafori* o *monitor*

- **Message passing**: nel modello message-passing i diversi task comunicano attraverso scambio di messaggi. La comunicazione può essere di tipo sincrono o asincrono.

- **Implicito**: nel modello implicito il meccanismo di interazione tra processi non è visibile al programmatore, il compilatore e/o il sistema runtime sono responsabili della parallelizzazione, questo è il caso di alcuni linguaggi specifici tipo *MATLAB*.

e alla **decomposizione del problema**:

- **Parallelismo a livello di task**: in questo modello si hanno diversi thread/processi (o flussi di esecuzione) assegnati a diversi processori, i thread/-processi possono eseguire lo stesso codice ma anche codice diverso e possono comunicare tra di loro attraverso i meccanismi di interazione esposti in precedenza

- **Parallelismo a livello di dati**: in questo modello diversi thread/processi eseguono lo stesso set di istruzioni su dati diversi, questi dati vengono in genere strutturati in array o matrici e sono divisi tra i diversi thread/processi i quali lavorano separatamente sulla propria porzione di dati.

3.5 Le prestazioni di un programma parallelo

Quando si progetta un programma parallelo è necessario tenere in considerazione diversi aspetti al fine di ottenere un programma efficiente:

- il tipo di comunicazione che può essere sincrona o asincrona: in caso di comunicazione sincrona si possono avere latenze indotte dall'interruzione dei processi in attesa di completamento della comunicazione

  campo della comunicazione che può essere punto a punto o collettiva. Più nello specifico le comunicazioni collettive possono essere di tipo broadcast, scatter o gather:

  - broadcast: comunicazione di tipo uno a molti in cui uno stesso dato viene inviato da un singolo processo a molti processi
  - scatter: comunicazione di tipo uno a molti in cui un processo invia diversi dati a diversi processi
  - gather: l'inverso della scatter, diversi dati sono presi da diversi processi e vengono "convogliati" in un singolo processo

- l'architettura su cui il programma andrà eseguito (memoria condivisa o distribuita, la piattaforma hardware, il tipo di interconnessione della rete) può condizionarne l'efficienza

Per testare la buona qualità di una parallelizzazione viene utilizzato un parametro chiamato **speedup**.
Lo speedup S_n è il rapporto tra il miglior tempo di esecuzione T_1 di un programma sequenziale e il tempo di esecuzione T_n dell'algoritmo parallelizzato eseguito su n processori.

$$S_n = \frac{T_1}{T_n}$$

Un programma parallelo è "ottimo" se presenta uno speedup lineare ovvero all'aumentare del numero di processori aumenta linearmente anche lo speedup, in pratica questo risultato è difficilmente raggiungibile.

In affiancamento allo speedup vengono spesso utilizzati altri due parametri, l'**efficienza** e la **scalabilità**.
L'efficienza misura quanto bene l'algoritmo parallelo sfrutti i processori sui quali è eseguito.

$$E_n = \frac{S_n}{n}$$

La scalabilità misura la capacità dell'algoritmo parallelo di rimanere efficiente all'aumentare del numero di processori. Esistono due diverse nozioni di scalabilità:

- **Scalabilità forte**:
  definita come la variazione del tempo di esecuzione dell'algoritmo al variare del numero di processori dato un problema di dimensione totale fissata

- **Scalabilità debole**:
  definita come la variazione del tempo di esecuzione dell'algoritmo al variare del numero di processori dato un problema di dimensione per processore fissata

Capitolo 4

L'algoritmo: dINSCY

In questo capitolo spiegheremo nel dettaglio l'algoritmo implementato: **dINSCY**.
Dapprima verranno spiegate e motivate alcune scelte progettuali intraprese come il linguaggio e le librerie utilizzate per l'implementazione.
Si passerà quindi a mostrare dapprima le strutture dati implementate e infine nel dettaglio l'algoritmo vero e proprio.

4.1 Le scelte progettuali

Il linguaggio scelto per l'implementazione dell'algoritmo è stato C++ principalmente per due motivazioni:

- un maggiore controllo a basso livello delle risorse e quindi una possibile maggiore efficienza sia in termini di prestazioni che in termini di utilizzo di memoria rispetto a linguaggi più ad alto livello
- utilizzo della libreria di message passing MPI per l'implementazione della parte di comunicazione/sincronizzazione tra processi

In particolare per quanto riguarda il secondo punto si è preferito implementare l'algoritmo in C++ in quanto, nonostante esistano binding che permettono di usare MPI su diversi linguaggi la libreria è nativamente implementata in C++ e Fortran (attraverso OpenMPI).
Facendo un passo indietro la prima scelta che si è dovuta prendere è stata quale paradigma di calcolo parallelo usare per l'implementazione di dINSCY; la scelta è ricaduta sul paradigma di calcolo distribuito per le seguenti motivazioni:

- possibilità di utilizzo di cluster di elaboratori per il testing dell'algoritmo contro la necessità di una macchina specifica con molti core e una memoria condivisa in caso di utilizzo del paradigma di calcolo parallelo
- un basso accoppiamento tra processi, su questo punto torneremo più avanti

La progettazione di dINSCY è stata fatta partendo dal lavoro di INSCY [2], le strutture dati e l'algoritmo in esso contenuti sono stati implementati e poi, tramite MPI si è reso l'algoritmo distribuito.
Passeremo ora a spiegare dINSCY, descrivendo dapprima le strutture dati utilizzate dall'algoritmo e le motivazioni che ne hanno portato alla scelta per poi passare a spiegare l'implementazione dell'algoritmo vero e proprio di subspace clustering. Infine andremo ad approfondire la parte di distribuzione dell'algoritmo e la comunicazione e sincronizzazione tra i processi.

4.2 Le strutture dati

La prima caratteristica di INSCY è quella di inserire gli oggetti in una **griglia a conservazione di densità** ovvero una griglia che divide ogni dimensione in un certo numero di intervalli e che presenta al bordo superiore di ogni intervallo una regione di larghezza ϵ definita bordo di connettività.
In dINSCY la griglia a conservazione di densità viene implementata da una serie di oggetti di tipo *UniDimensionalGrid*; ogni oggetto rappresenta la griglia in una specifica dimensione, definisce il numero delle celle e la loro dimensione oltre che la dimensione dei bordi di connettività tra celle. La griglia non contiene veramente gli oggetti del dataset ma mette a disposizione una serie di metodi che permettono di ricavare la posizione di ogni oggetto in tempo reale, ovvero l'intervallo in cui il punto va a posizionarsi in ogni dimensione, questi metodi sono:

computeCellNumber(Object o, int d): dato un oggetto o e una dimensione d restituisce l'intervallo in cui l'oggetto è posizionato nella dimensione d

isAtBorder(Object o, int d): dato un oggetto o e una dimensione d restituisce true se l'oggetto si trova nel bordo di una cella, false altrimenti

Gli oggetti sono rappresentati in dINSCY da oggetti di tipo *Object* che contengono:

id: identificatore dell'oggetto

attributes: vettore degli attributi dell'oggetto

Gli oggetti vengono fisicamente memorizzati in un oggetto di tipo *DataBase*, il quale memorizza, oltre che gli oggetti veri e propri alcune informazioni utili a dINSCY come l'attributo massimo e minimo tra tutti gli oggetti per ogni dimensione.

La principale struttura dati utilizzata da dINSCY è lo SCY-tree. Grazie allo SCY-tree è possibile effettuare una efficiente indicizzazione dello spazio; ricordiamone brevemente la definizione:

- albero che definisce una restrizione dello spazio originario in cui ogni nodo è un descrittore (d, i) con d dimensione e i intervallo
- identificato dalla sequenza di restrizioni dello spazio che lo hanno originato (l'albero dello spazio intero ha descrittore nullo)
- ogni nodo è inserito in una lista concatenata che permette di accedere velocemente a tutti i nodi aventi lo stesso descrittore
- tutti i nodi sono ordinati lessicograficamente, prima sulla dimensione e poi sull'intervallo

In dINSCY lo SCY-tree è implementato dalla classe *ScyTree*. Questa contiene il descrittore dell'albero, un contatore degli oggetti nella regione di spazio e il puntatore al nodo radice.
Il descrittore dell'albero è un insieme ordinato di elementi di tipo *TreeID*. Ogni elemento *TreeId* indica una specifica restrizione dello spazio attraverso la dimensione e l'insieme di intervalli considerati.
I nodi dell'albero sono oggetti di tipo *ScyNode* ognuno dei quali rappresenta i punti in una specifica regione dello spazio.
Uno ScyNode consiste in un *Id* (dimensione, intervallo, bordo) che indica una specifica regione di una specifica dimensione dello spazio, un contatore degli oggetti nella regione di spazio rappresentata dall'albero radicato in questo nodo, un puntatore allo *ScyNode* padre e un insieme di puntatori agli *ScyNode* figli.

I nodi si distinguono in normali o di bordo a seconda del parametro *bordo* di *Id*.
Le linked-list e l'ordinamento lessicografico dei nodi è stato implementato tramite

una unica struttura dati, una mappa che ha come chiavi gli *Id* dei nodi e come valori vettori di nodi con *Id* uguale alla chiave; ogni elemento della mappa quindi contiene tutti i nodi con stesso identificatore contenuti nell'albero.
L'ordinamento lessicografico è implicito nella definizione di mappa.
Vediamo ora un esempio di SCY-tree (sulla sinistra) e la relativa struttura dati *ScyTree* di dINSCY (sulla destra):

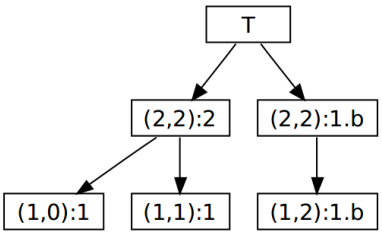


Figura 4.1: SCY-tree

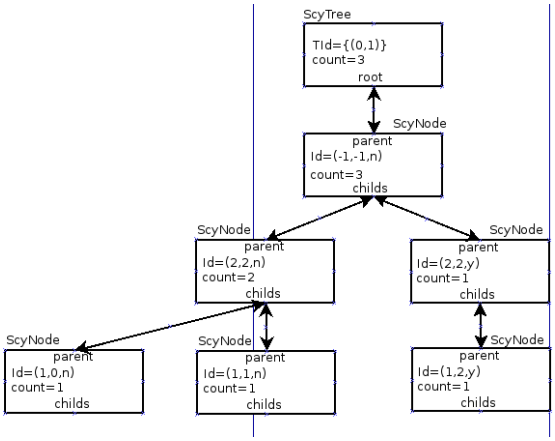


Figura 4.2: Struttura dati ScyTree

Come possiamo vedere nella figura 4.2 abbiamo un oggetto che rappresenta lo ScyTree con identificatore *TId* e un puntatore al nodo radice, abbiamo poi un oggetto per ogni nodo dello ScyTree, ogni oggetto *ScyNode* contiene l'identificatore *Id* del nodo, il contatore *count* degli oggetti presenti nella regione di spazio identificata dal nodo e la variabile *border* che indica se il nodo rappresenta o no punti al bordo (*border*). Ogni nodo ha poi un puntatore al nodo padre e un insieme di puntatori ai nodi figli. Vediamo ora la mappa corrispondente a questo SCY-tree:

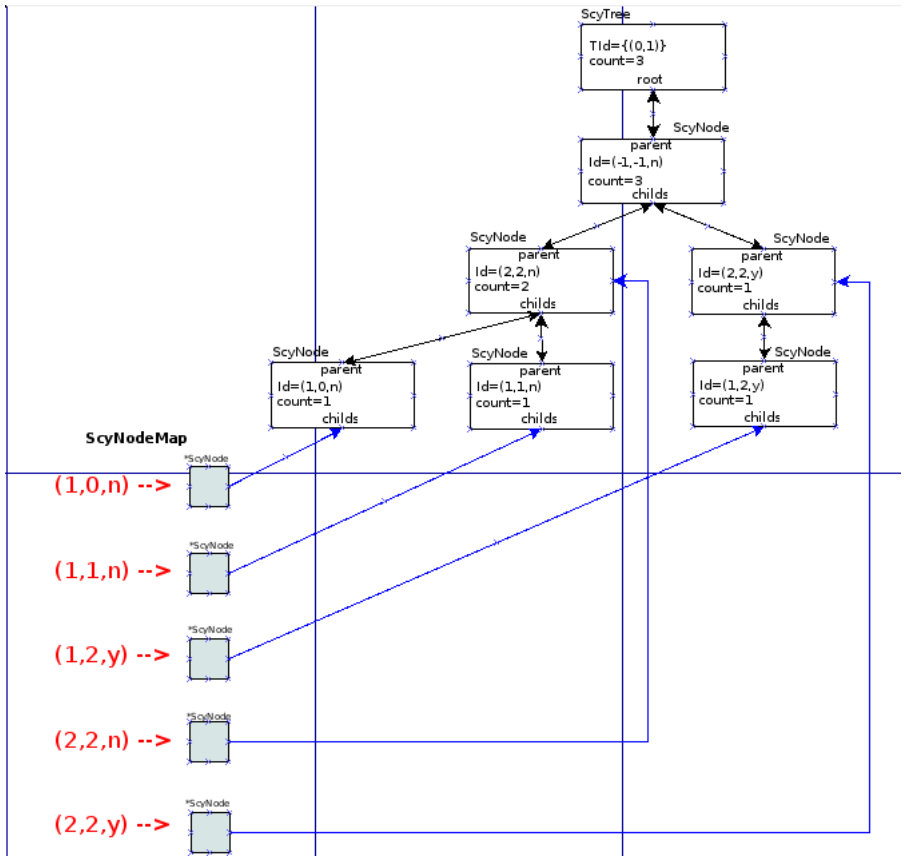


Figura 4.3: Mappa dei nodi uguali nello ScyTree

Come si può vedere dalla figura 4.3 la mappa ha come chiavi gli id dei nodi e ogni chiave è associata a un vettore di puntatori ai nodi dell'albero (frecce in blu), in questo modo si può accedere velocemente a nodi uguali nell'albero senza la necessità di cercarli nello stesso.

I cluster trovati vengono memorizzati in una mappa indicizzata sull'identificatore dello *ScyTree* da cui i cluster originano.

Per la parte di parallelizzazione è stata implementata una struttura dati ad-hoc atta a memorizzare i messaggi inviati/ricevuti: *MpiData*.
Questa struttura dati contiene:

un parametro ***die*** che indica che il messaggio riguarda la fine dell'esecuzione del processo ha inviato il messaggio

un insieme ***clusters*** che contiene i cluster che si stanno inviando

I messaggi inviati dai processi sono di due tipi:

- **fine dell'esecuzione**: il processo comunica che ha terminato l'esecuzione: $die = true \wedge clusters = \emptyset$
- **cluster trovato/i**: il processo comunica dei cluster trovati: $die = false \wedge clusters \neq \emptyset$

Inoltre viene usata un vettore di booleani che memorizza i processi ancora attivi nel comunicatore.

4.3 I parametri e il file di input

dINSCY prende alcuni parametri in input, questi sono:

- ***epsilon***: la larghezza del bordo di ogni cella nella griglia a conservazione di densità
- ***gridsize***: la larghezza di ogni cella della griglia a conservazione di densità
- ***minpoints***: il numero minimo di punti di ogni cluster
- ***minsize* e *density***: definiscono la soglia di densità di ogni cluster

dINSCY accetta in input file nel formato *.arff* (Attribute-Relation File Format).
Un file *.arff* è composto da due differenti sezioni, una parte di **intestazione** e una parte **dati**. La parte **intestazione** contiene:

nome della relazione, indicata con @relation [nome]

l'elenco degli attributi della relazione, ogni attributo è indicato con @attribute [nome] [tipo] con tipo:

- numeric: numeri reali/interi (es. @attribute number numeric)
- nominal-specification: elenco dei possibili valori nominali (es. @attribute class valA, valB, valC, valD)
- string: valore testuale (es. @attribute text string)
- date: data, la dichiarazione di questo tipo è nella forma @attribute [nome] date [formato] con formato il formato della data secondo lo standard ISO-8601 (es. @attriute timestamp date yyyy-MM-dd HH:mm:ss)

La parte **dati** inizia con la dichiarazione *@data* e contiene un insieme di righe, una per ogni oggetto e ogni riga contiene gli attributi separati da una virgola.

Di seguito un esempio di file *.arff*, una parte del dataset **Iris-setosa**:

```
@relation iris-setosa

@attribute sepallength numeric
@attribute sepalwidth numeric
@attribute petallength numeric
@attribute petalwidth numeric
@attribute class Iris-setosa,Iris-versicolor,Iris-virginica

@data

5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

4.4 L'implementazione

L'algoritmo di clustering di dINSCY segue l'algoritmo INSCY (spiegato in 2.3.1) integrandolo delle opportune parti di distribuzione del lavoro, sincronizzazione e comunicazione tra processi rendendolo di fatto una versione distribuita di INSCY.

Spiegheremo l'algoritmo prendendo ogni fase dell'algoritmo INSCY e spiegando come questa sia stata implementata in dINSCY, partendo dalla lettura del file di

input fino ad arrivare alla stampa dei risultati dell'algoritmo.
Prima di spiegare l'algoritmo vero e proprio è necessario arrivare a costruire le strutture dati su cui l'algoritmo andrà a lavorare ovvero il *Database*, lo *ScyTreee* e la mappa dei nodi uguali.

4.4.1 La costruzione del database

La lettura del file di input e il popolamento del database sono affidati alla classe *ArffDatabase*, per la lettura del file di input ci si è affidato ad una libreria esterna di parsing specifica per il formato *.arff*: *ArffReader Library*. Questa libreria consta di un oggetto di tipo *ArffParser* che contiene il file da parsare; attraverso una chiamata al metodo *parse()* il file viene interamente analizzato e il risultato restituito in un oggetto di tipo *ArffData* dal quale è possibile accedere al nome della relazione, alla sezione di header e alla sezione dati. Per accedere alla sezione *@data* contenente la dichiarazione degli oggetti del dataset è necessario chiamare il metodo *get_instance()* dell'oggetto di tipo *ArffData*, viene cosi restituito un oggetto di tipo *ArffIstance* il quale contiene uno degli oggetti del dataset. Infine un oggetto di tipo *ArffValue* conterrà il singolo attributo dell'oggetto.
Man mano che gli oggetti vengono letti sono inseriti nell'oggetto di tipo *DataBase* deputato a contenere il dataset e alla fine dell'inserimento viene calcolata per ogni dimensione l'attributo minimo e massimo tra tutti gli oggetti del dataset.
Ricapitolando i passi per la lettura del file e creazione del database di dINSCY sono:

- ***ArffDatabase db = new ArffDataBase();***
- ***ArffParser parser = new ArffParser(path/file.arff);***
- ***ArffData data = parser.parse();***
- per ogni oggetto di indice i del dataset (il numero degli oggetti è uno degli attributi dell'oggetto *data*)
  - ***ArffInstance instance = data.get_instance(i);***
  - ***Object obj = new Object(i);***

- per ogni attributo di indice j dell'oggetto (il numero degli attributi può essere richiamato dall'oggetto *instance*)
  - ***ArffValue value = instance.get(j);***
  - ***obj.insertAttribute(value);***
- ***db.insertObject(obj);***

- ***db.computeMinMaxAttributes();***

4.4.2 La costruzione dello SCY-tree

Una volta riempito il database con i dati contenuti nel file arff va costruito lo SCY-tree sul quale successivamente andrà a lavorare l'algoritmo. Dapprima viene inizializzata la griglia a conservazione di densità ovvero viene creato un oggetto di tipo *UniDimensionalGrid* passandogli come parametri di inizializzazione *gridsize* e *epsilon*:

- gridsize: dimensione di ogni cella
- dimensione del bordo tra celle

La dimensione della griglia (numero di celle) viene ricavata a partire dal database: per ogni dimensione d, viene calcolato il numero di celle necessarie per contenere gli oggetti prendendo il valore massimo MAX e il valore minimo MIN degli attributi degli oggetti per ogni dimensione d, sottraendoli tra di loro e dividendone il risultato per *gridsize* ottenendo cosi un vettore contenente per ogni dimensione d il numero di celle necessarie a contenere le proiezioni degli oggetti in d.
Prendendo il valore massimo di questo vettore si ottiene il numero di celle che la griglia dovrà avere e questo diventa il numero di celle che avrà la griglia (ricordiamo che una caratteristica della griglia è l'avere stesso numero e larghezza di celle in tutte le dimensioni).
Passiamo ora a spiegare come lo SCY-tree viene creato a partire dal database e dalla griglia.
La creazione dello SCY-tree viene fatta scorrendo, per ogni oggetto tutti i suoi attributi, e per ogni attributo andando a creare (o aggiornare) lo SCY-node corretto; l'inserimento (gli oggetti non vengono veramente inseriti ma vengono usati per la creazione dei nodi dell'albero) di un oggetto corrisponde a una visita in profondità di un ramo dello SCY-tree. Vediamo ora più in dettaglio i passi fatti per la creazione dello SCY-tree:

- si mantengono due puntatori ai nodi dell'albero, **pnode** e **cnode**
- Per ogni oggetto **obj** del database:
  - L'inserimento di un oggetto inizia sempre dalla radice dell'albero:
    pnode = cnode = root;
  - Per ogni dimensione **d** di *obj*
    - si calcola l'intervallo in cui ricade l'oggetto *obj* nella dimensione *d*, per fare questo si utilizzano le funzioni messe a disposizione dall'oggetto di tipo *UnidimensionalGrid*:
      interval = grid.computeCellNumber(obj,d);
    - si calcola se l'oggetto *obj* ricade o meno nel bordo dell'intervallo, anche per fare questo si utilizzano le funzioni messe a disposizione dall'oggetto di tipo *UnidimensionalGrid*:
      border = grid.isAtBorder(obj,d);
    - si inserisce/aggiorna il nodo nello SCY-tree:
      cnode = pnode.addChildNode(obj,j,interval,border);
      - si cerca tra i figli di pnode se esiste un nodo con ***ID = (interval, j, border)***
      - se presente lo si aggiorna ***node.counter++;***
      - se non presente si crea un nuovo nodo con ***counter = 1*** e lo si inserisce tra i figli di pnode
      - viene restituito il nodo aggiornato/creato
    - si inserisce (se non già presente) il nodo dello SCY-tree nella mappa dei nodi uguali
      map.addNode(cnode);

Ora che lo SCY-tree è stato creato dINSCY è pronto per andare alla ricerca dei cluster, per spiegarne il funzionamento divideremo l'algoritmo in fasi, sono le stesse fasi spiegate in 2.3.1 intercalate, dove necessario, da operazioni di sincronizzazione e comunicazione tra processi.

4.4.3 L'algoritmo

- **FASE MPI 0: dataDistribution()**
  Nella fase iniziale viene effettuata la distribuzione dei dati tra i processi

MPI, per capirne il funzionamento esponiamo dapprima l'idea che sta dietro all'esecuzione distribuita di dINSCY partendo da come funziona INSCY.
Nella fase 1 di INSCY si effettua una restrizione su un nodo dello SCY-tree: viene generato un nuovo SCY-tree sul quale si procede con le fasi successive. Caratteristica importante è che in ognuna di queste fasi si utilizzano solo ed esclusivamente lo SCY-tree ristretto e lo SCY-tree da cui quello ristretto è stato generato, non vengono utilizzati quindi SCY-tree ristretti in altre dimensioni, questa caratteristica è alla base dell'esecuzione distribuita di dINSCY.
L'idea è questa: supponiamo di avere p processi e n nodi, distribuiamo all'inizio dell'esecuzione gli n nodi dello SCY-tree iniziale (creato a partire dal dataset) tra i p processi in parti bilanciate. Facciamo quindi eseguire ad ogni processo l'algoritmo con i soli nodi a lui assegnati. Avremo quindi p processi ogniuno dei quali esegue inizialmente n/p restrizioni e questi processi, grazie alla caratteristica esposta poco sopra, dovranno comunicare esclusivamente per mantenere aggiornate a livello globale le strutture dati contenenti i cluster trovati durante l'esecuzione dell'algoritmo.

Nella fase 0 quindi si va a distribuire i nodi dello SCY-tree tra i processi, questa in realtà non è una vera e propria distribuzione ma bensi un calcolo effettuato da ogni processo dei nodi che dovrà utilizzare. Tutti i processi all'inizio dell'algoritmo leggono il file di input e creano tutte le strutture dati, di fatto ogni processo avrà una copia dello SCY-tree intero, caratteristica questa fondamentale per eseguire le restrizioni.

```
1         int div[parameters->mpiSize];
2         int d = std::floor(map.size() / parameters->mpiSize);
3         for (int i = 0; i < parameters->mpiSize; ++i) {
4             div[i] = d;
5         }
6         int rest = map.size() % parameters->mpiSize;
7         for (int i = 0; i < rest; ++i) {
8             int index = i % parameters->mpiSize;
9             ++div[index];
10        }
11        //set start and stop index
12        parameters->startIndex[0] = 0;
13        parameters->endIndex[0] = div[0] - 1;
14
15        int start = 0;
16        for (int i = 0; i < parameters->mpiRank; ++i) {
17            start += div[i];
18        }
19        int end = start + div[parameters->mpiRank] - 1;
20        parameters->startIndex[parameters->mpiRank] = start;
21        parameters->endIndex[parameters->mpiRank] = end;
```

Listato 4.1: Distribuzione nodi SCY-tree tra processi

Spieghiamo ora questo pezzo di codice:

- Si utilizza un vettore *div* che contiene tanti interi quanti sono i processi, l'intero di indice i all'interno del vettore indica il numero di nodi assegnato al processo i-esimo
- Nelle righe da 1 a 5 vengono distribuiti in parti uguali i nodi dello SCY-tree tra i processi. Si noti che i nodi vengono in realtà contati dalla mappa dei nodi e non direttamente dallo SCY-tree, questo in quanto lo SCY-tree contiene ovviamente nodi che sono ripetuti mentre ai processi vanno distribuiti i nodi senza tener conto del numero di volte in cui compaiono nell'albero
- Nelle righe da 6 a 10 gli eventuali nodi rimasti da distribuire vengono assegnati uno ad uno ai processi con politica round-robin fino ad esaurirli tutti. A questo punto il vettore div contiene effettivamente per ogni processo il numero di nodi su cui deve eseguire l'algoritmo
- Nelle righe da 12 a 19 ogni processo calcola gli indici all'interno della mappa dei nodi del primo e dell'ultimo nodo su cui dovranno andare ad operare, questo è possibile in quanto ogni processo, oltre che essere a conoscenza del numero di nodi su cui dovrà lavorare (avendolo calcolato precedentemente) conosce anche il suo rango ovvero la sua posizione all'interno del comunicatore MPI e quindi quanti processi con rango inferiore al suo ci sono nel comunicatore.

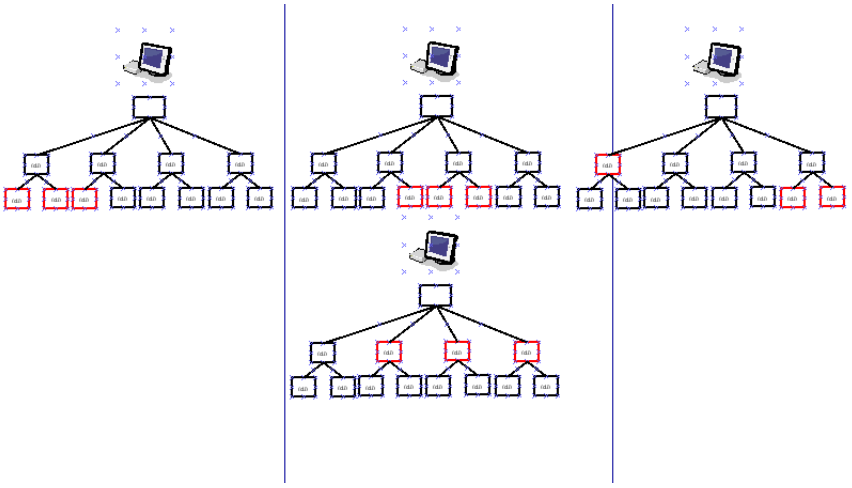


Figura 4.4: Processi e dati

In figura 4.4 possiamo vedere una rappresentazione di 4 processi MPI, ogni uno dei quali possiede una sua copia dell'intero SCY-tree (costituito da 12 nodi).
In rosso sono indicati i nodi distribuiti (calcolati) allo specifico processo.
Completata questa fase ogni processo può passare alla fase successiva. Fino alla prossima fase di comunicazione i processi saranno perfettamente indipendenti.

- **FASE 1: restrizione SCY-tree**
  La parte di restrizione dello SCY-tree viene implementata nella funzione ***restrictTree(ScyTree *t, ID id)***: preso uno ScyTree t e l'ID del nodo su cui restringere restituisce lo ScyTree ristretto.
  Ricordiamo che la restrizione di uno SCY-tree su un nodo (d, i) consiste in due fasi distinte: eliminazione di tutti i rami dell'albero che non contengono il nodo (d, i) e eliminazione del livello dell'albero corrispondente alla dimensione d.

Vediamo dapprima la prima fase di restrizione:

```
1         for (int interval = 0; interval <= parameters->grid->getSize(); ++interval) {
2             if (interval != id.interval) {
3                 ID newId(id.dimension, interval, id.border);
4                 it = map.find(newId);
5                 if (it != map.end()) {
6                     for (std::vector<ScyNode *>::iterator it2 = it->second.begin(); it2 != it->
                          second.end(); ++it2) {
7                         ScyNode *n = (*it2);
8                         tc->removeNode(n, n);
9                     }
10                }
11            }
12        }
```

Listato 4.2: Funzione restrictTree(): eliminazione rami

  - Nelle righe da 1 a 3 vengono generati gli ID con dimensione d e intervallo diverso da i (si accede alla griglia a conservazione di densità per scorrere gli intervalli)
  - Nella riga 4 si cerca nella mappa dei nodi la chiave appena generata
  - Se questa viene trovata si accede allo *ScyTree* attraverso il vettore di puntatori associato alla chiave trovata e si vanno ad eliminare i rami dell'albero che contengono i nodi trovati (righe da 5 a 8)

Vediamo ora la seconda fase di restrizione:

```
1          it = map.find(id);
2
3          if (it != map.end()) {
4              v = &(it->second);
5
6              for (std::vector<ScyNode *>::const_iterator it = v->begin(); it != v->end(); ++it) {
7                  ScyNode *me = (*it);
8
9                  ScyNode *dad = me->getParentNode();
10                 ScyNode *sun;
11
12                 //erase me from my dad childs
13                 dad->getChildsNodes().erase(me->getId());
14
15                 for (std::map<ID, ScyNode *>::iterator it2 = me->getChildsNodes().begin(); it2 != me
                        ->getChildsNodes().end(); ++it2) {
16                     sun = (*it2).second;
17                     //point parent to my dad
18                     sun->setParent(dad);
19                     //add my child to my parent childs
20                     dad->getChildsNodes()[sun->getId()] = sun;
21                 }
22
23
24             }
25         }
```

Listato 4.3: Funzione restrictTree(): eliminazione livello

Notiamo che nella fase precedente sono stati rimossi tutti i rami che non contengono il nodo ristretto, quindi a livello d sono rimasti solo nodi uguali a quello ristretto, eliminare il livello d è quindi equivalente ad andare ad eliminare i nodi ristretti.

- Nella riga 1 si cerca la chiave $ID = (d, i)$ nella mappa dei nodi
- Se questa viene trovata si accede allo *ScyTree* attraverso il vettore di puntatori associato alla chiave trovata e dapprima si elimina il nodo dai figli del padre (righe da 7 a 13)
- Infine si vanno a collegare i figli del nodo eliminato al padre (righe da 15 a 20)

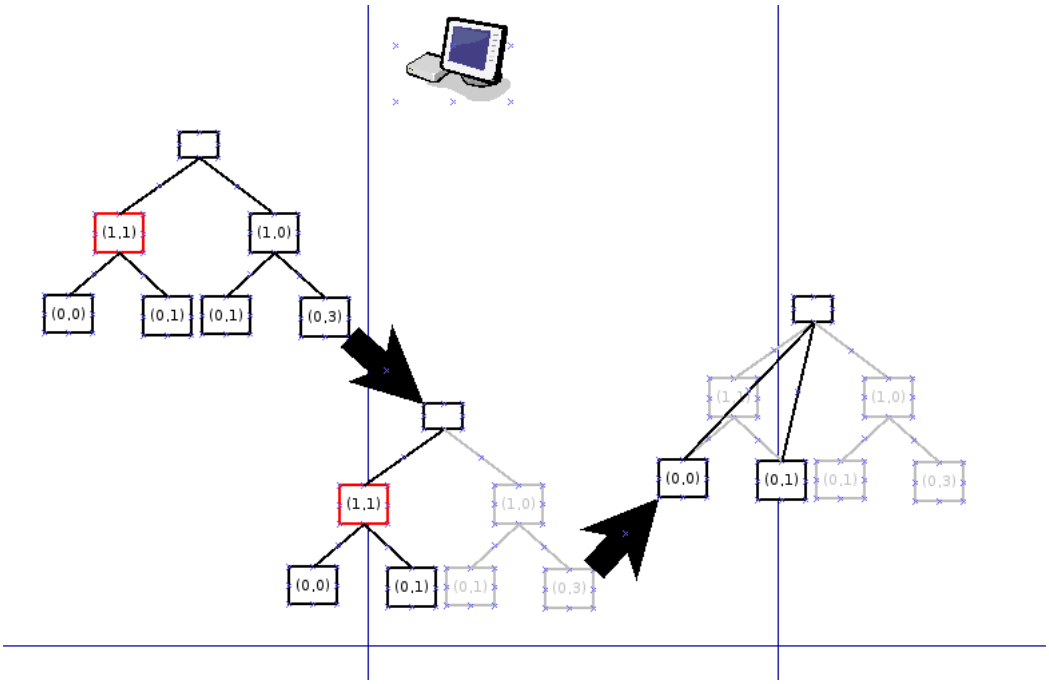


Figura 4.5: Esempio restrizione

In figura 4.5 possiamo vedere, in ordine, le due fasi della restrizione dello SCY-tree sul nodo $(1, 1)$.

- **FASE 2: unione SCY-tree**
  La seconda fase dell'algoritmo è quella di ricerca e unione degli SCY-tree "vicini", questa fase è implementata nella funzione ***mergeWithNeighbors(ScyTree t,resTree, Map map, ScyNode n)*** la quale, preso lo SCY-tree ristretto, l'albero e il nodo di origine e la mappa dei nodi di t, cerca eventuali possibili unioni tra SCY-tree, le effettua e restituisce lo SCY-tree risultante.

```
1          std::vector<ID> toMerge;
2          ID id = n->getId();
3          while (true) {
4              id.border = true;
5              //search for s-connector
6              std::map<ID, std::vector<ScyNode *> >::iterator it = map.find(id);
7              if (it != map.end()) {
8                  //s-connector found, search for next interval for merge
9                  id.border = false;
10                 ++id.interval;
11                 std::map<ID, std::vector<ScyNode *> >::iterator it2 = map.find(id);
12                 if (it2 != map.end()) {
13                     //found point in next interval, add to merge list
14                     toMerge.push_back(id);
```

```
15                  } else {
16                      break;
17                  }
18              } else {
19                  break;
20              }
21          }
22          return performMerges(restricted_tree, t, map, toMerge);
```

Listato 4.4: Funzione mergeWithNeighbors()

Si noti che l'albero *resTree* può essere unito ad altri alberi se nell'albero da cui origina (t) esiste un nodo s-connettore che connette n ad un altro nodo. L'unione consiste nel cercare questi nodi s-connettori e se questi esistono procedere a unire gli alberi ristretti.

- Nella riga 6 viene cercato il nodo s-connettore nella mappa dei nodi di t
- Se il nodo s-connettore viene trovato si procede a cercare l'eventuale nodo connesso ad n tramite questo (righe da 9 a 11)
- Se il nodo viene trovato viene aggiunto al vettore dei merge da effettuare
- Una volta analizzati tutti gli s-connettori trovati si effettuano i merge, l'operazione di merge tra due alberi t1 e t2 consiste semplicemente nell'inserire i rami di t1 in t2, sommando eventualmente i contatori di nodi uguali

- **FASE 3: pruning ricorsione**
  Il pruning della ricorsione consiste nel verificare se il numero di oggetti rappresentati dallo SCY-tree è o meno superiore a *minSize* (parametro di input).
  Se questo non è vero l'albero viene potato, una ulteriore restrizione porterebbe certamente ad un albero con numero di oggetti rappresentati inferiore.

- **FASE MPI 1: broadcastReceive()**
  Questa è la prima fase di comunicazione tra processi.Come abbiamo detto i processi comunicano solo per tenere sincronizzate le strutture dati contenenti i cluster trovati. Questo si rende necessario per la FASE 4 dell'algoritmo in cui si va a verificare la ridondanza dello SCY-tree (cluster potenziale) rispetto ai cluster già trovati; risulta chiaro che un insieme di cluster trovati

incompleto potrebbe portare a risultati errati.

La funzione ***broadcastReceive()*** è deputata a ricevere tutti i messaggi pendenti da tutti i processi, per fare questo utilizza un meccanismo asincrono di tipo **probe and receive**.

I messaggi ricevuti vengono inseriti nella struttura dati *MPIData* e a seconda del tipo di messaggio viene intrapresa l'azione appropriata.

```
1         MpiData inMessage;
2         //receive messages
3         int retValue = 0;
4         while (boost::optional<boost::mpi::status> status = parameters->mpiWorld->iprobe(boost::mpi
              ::any_source, boost::mpi::any_tag)) {
5             //clean message
6             inMessage.die = false;
7             inMessage.cluster.clear();
8             inMessage.id.clear();
9             //recv probed message
10            parameters->mpiWorld->irecv(status->source(), status->tag(), inMessage);
11            if (inMessage.die) {
12                parameters->liveProcess[status->source()] = false;
13            }
14            if (inMessage.cluster.size() != 0) {
15                retValue++;
16
17                //from intCluster to objCluster
18                Clustering::ObjClustersCollection objClusters;
19                int i = 0;
20                for (std::vector<Clustering::IntCluster>::iterator it = inMessage.cluster.begin();
                       it != inMessage.cluster.end(); ++it) {
21                    Clustering::ObjCluster objCluster;
22                    for (Clustering::IntCluster::iterator it2 = it->begin(); it2 != it->end(); ++it2
                           ) {
23                        unsigned int objID = *it2;
24                        objClusters[i].insert(parameters->db->getObject(objID));
25                    }
26                    ++i;
27                }
28
29                Clustering::ClustersCollection::iterator item = dimensionFindInClusters(clusters,
                       inMessage.id);
30                if (item != clusters.end()) {
31                    Clustering::ObjClustersCollection *localClusters = &item->second;
32                    unsigned int last_index = localClusters->rbegin()->first;
33                    //check if cluster already exist
34                    for (Clustering::ObjClustersCollection::iterator it = objClusters.begin(); it !=
                             objClusters.end(); ++it) {
35                        bool found = false;
36                        for (Clustering::ObjClustersCollection::iterator it2 = localClusters->begin
                               (); it2 != localClusters->end(); ++it2){
37                            if (compareClusters(it->second, it2->second)){
38                                found = true;
39                                break;
40                            }
41                        }
42                        if (!found){
43                            ++last_index;
44                            (*localClusters)[last_index] = it->second;
45                        }
46
47                    }
48                } else {
49                    clusters[inMessage.id] = objClusters;
50                }
51            }
52        }
```

```
53          return retValue;
```

Listato 4.5: Funzione broadcastReceive()

- Nella riga 5 viene effettuato il probe dei messaggi da tutti i processi
- Fino a quando c'è un messaggio da un processo p in attesa di essere ricevuto:
  - Si riceve il messaggio da p (riga 11)
  - Se il messaggio è un messaggio di **fine dell'esecuzione** viene aggiornata la struttura dati preposta alla memorizzazione dei processi ancora attivi per rendere inattivo il processo da cui si è ricevuto il messaggio (riga 13)
  - Se il messaggio è un messaggio di **cluster trovato** allora il cluster viene inserito nell'insieme dei cluster trovati

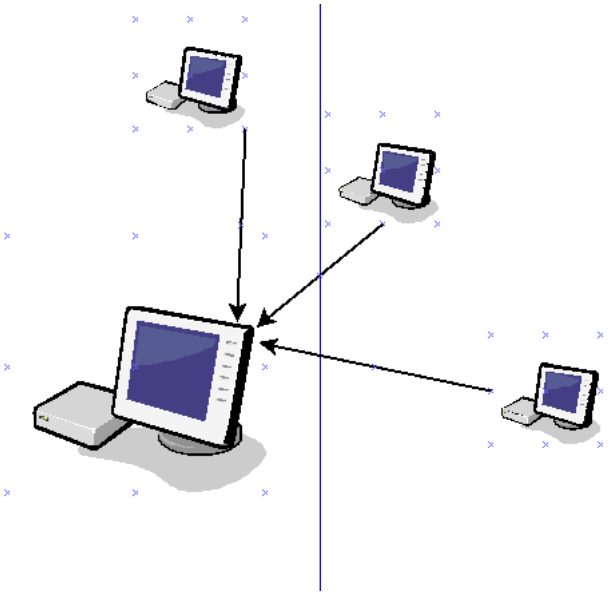


Figura 4.6: Ricezione messaggi da tutti i processi

- **FASE 4: in-process removal**
  In questa fase viene eseguito il pruning delle ridondanze, ovvero si verifica se esistono cluster che coprono parzialmente o totalmente il potenziale cluster che potrebbe originare dall'albero ristretto in analisi.

```
1          std::vector<int> intCluster;
2          std::vector<TreeID> regionID = *restricted_tree->getID();
3          for (Clustering::ClustersCollection::iterator it = clusters.begin(); it != clusters.end();
                ++it) {
4              std::vector<TreeID> clusterID = it->first;
5              for (Clustering::ObjClustersCollection::iterator it2 = it->second.begin(); it2 != it->
                    second.end(); ++it2) {
6                  unsigned int clusterSize = it2->second.size();
7                  unsigned int regionSize = parameters->redundancy_factor * restricted_tree->
                        getCounter();
8                  if (containsID(clusterID, regionID) && (clusterSize >= regionSize)) {
9                      return true;
10                 }
11             }
12         }
13         return false;
```

Listato 4.6: Fase di in-process removal

Nelle righe 3-6 si scorre l'insieme dei cluster trovati, per ogni cluster C si verifica se C è ridondante rispetto all'albero *restricted_tree*, nella riga 7 si moltiplica la dimensione dell'albero per il fattore di ridondanza; il fattore di ridondanza indica la percentuale di copertura che bisogna avere perchè un cluster sia considerato ridondante rispetto allo SCY-tree in analisi ($R = 1$ copertura piena). Nella riga 8 si effettua la verifica vera e propria, C è ridondante rispetto a *restricted_tree* se:

- l'albero è contenuto in un sottospazio dello spazio in cui è contenuto il cluster (l'id dello SCY-tree è contenuto nell'id del cluster)
- il cluster copre in una certa percentuale (rispetto a R) lo SCY-tree

In caso la verifica sia positiva l'albero viene potato evitando cosi di effettuarvi sopra una costosa computazione di densità alla ricerca di cluster.

- **FASE 5: clustering**
  I soli alberi non potati dalle fasi precedenti sono potenziali candidati a contenere cluster. In questa fase viene applicato l'algoritmo vero e proprio di clustering.
  Questo algoritmo lavora sui punti e non sull'albero, il primo passo effettuato è quello di ricostruire, a partire dallo SCY-tree il database degli oggetti, una volta ricostruito su questo viene applicato l'algoritmo di clustering; dINSCY utilizza ***DBSCAN*** per la ricerca dei cluster.
  I cluster individuati da *DBSCAN* vengono successivamente passati attraverso un filtro che tiene in considerazione della dimensionalità dello spazio, vengono cosi rimossi quegli oggetti dal cluster che non sono densi rispetto alla dimensionalità.

- **FASE MPI 2: broadcastSend()**
  Questa è la seconda fase di comunicazione di *dINSCY*, in questa fase, se vengono trovati cluster (o se si termina l'esecuzione) viene inviato un messaggio in broadcast a tutti i processi.

```
1          int i = 0;
2          for (std::vector<bool>::iterator it = parameters->liveProcess.begin(); it != parameters->
               liveProcess.end(); ++it) {
3              bool value = *it;
4              if (value) {
5                  parameters->mpiWorld->send(i, CLUSTER_MESSAGE, outMessage);
6
7              }
8              ++i;
9          }
```

Listato 4.7: Funzione broadcastSend()

Nella riga 2 viene scorso il vettore dei processi ancora vivi (*liveProcess*), in questo modo il messaggio viene inviato solo a processi che effettivamente eseguiranno una corrispondente *receive* e non si ha spreco di risorse.
L'invio viene fatto nella riga 5.

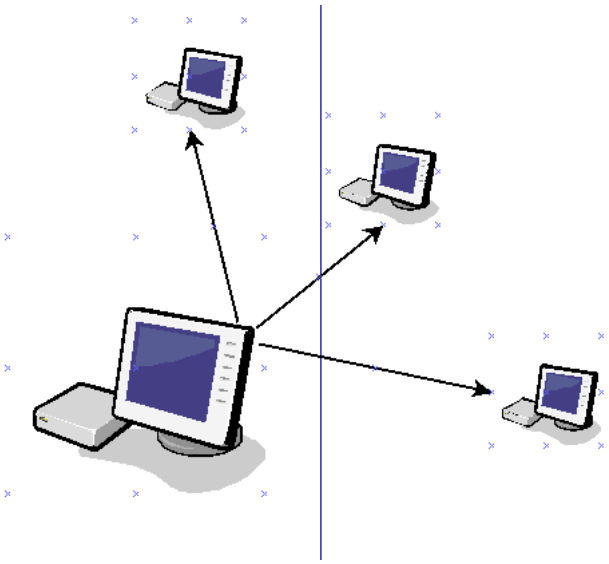


Figura 4.7: Invio in broadcast dei cluster trovati

Capitolo 5

Risultati sperimentali: prestazioni di dINSCY

In questo capitolo verrano mostrati i risultati di alcuni test prestazionali eseguiti su dINSCY.
Per misurare le prestazioni di dINSCY si è scelto di utilizzare come parametro la **scalabilità**. In particolare ètata scelta la nozione di **scalabilità forte** di cui diamo la definizione:

> **Scalabilità forte**:
> dato in input un dataset di dimensione fissa si va a misurare la variazione del tempo di esecuzione dell'algoritmo all'aumentare del numero di processori usati. L'obiettivo di questa tipologia di analisi è quello di trovare il numero giusto di processori che permettano di completare l'algoritmo in un tempo ragionevole senza sprecare cicli di clock a causa dell'overhead di comunicazione/sincronizzazione tra processi.

5.1 L'ambiente di esecuzione

L'ambiente di test utilizzato per dINSCY è un sottoinsieme delle macchine del cluster del dipartimento di informatica dell'università di Bologna.
In particolare sono state utilizzate 32 macchine collocate nei laboratori Ercolani e Ranzani.
Ogni macchina presenta la seguente configurazione hardware:

- processore dual-core con architettura i686

- 2 GB di memoria RAM

e la seguente configurazione software:

- distribuzione linux basata su Debian
- kernel linux versione 3.2.x
- OpenMPI versione 1.4.3 [1]
- libreria boost/mpi versione 1.46.1 [2]

Le macchine sono interconnesse tra di loro attraverso una rete gigabit ethernet (1000 Mbit).

5.2 Configurazione MPI

Il comando **mpirun** è uno script per l'esecuzione dei programmi MPI su diversi tipologie di sistemi, permette l'esecuzione "trasparente" sia su una singola macchina che su un cluster di macchine. In caso di esecuzione su cluster è necessario fornire allo script l'elenco delle macchine su cui dovranno essere lanciati i singoli processi MPI. Il file di configurazione utilizzato da mpirun per conoscere queste macchine si chiama **hostfile**. L'hostfile contiene, uno per ogni riga, gli indirizzi delle macchine con l'indicazione (facoltativa) del numero di slot di calcolo disponibili e il numero massimo di processi che possono essere lanciati su quella macchina:
IP/address [slots=x] [max-slots=y]
ESEMPIO:
localhost slots=2 max-slots=2
indica che la macchina con indirizzo localhost (ovvero la macchina da cui lo script mpirun viene lanciato) ha 2 slot di calcolo e vi si possono lanciare al massimo due processi.

Il comando mpirun accetta inoltre un altro parametro: il numero dei processi da eseguire.

[1] ultima versione di OpenMPI disponibile al momento della stesura di questo documento: 1.6.4

[2] ultima versione di boost/mpi disponibile al momento della stesura di questo documento: 1.53.0

Il parametro *-np X* indica che il numero di processi MPI da lanciare devono essere X, la strategia di lancio utilizzata da ORTE (Open MPI's run-time environment, il motore di esecuzione dei processi di MPI) è una strategia di tipo **by-slot**: i processi vengono schedulati su tutti gli slot disponibili di un nodo prima di passare al nodo successivo.
Oltre a questi due sono stati utilizzati altri due parametri:

- *–mca btl ^openib*:
  questo parametro indica di non caricare il modulo per lo stack OpenIB (per le reti Infiniband) utilizzando esclusivamente lo stack TCP
- *–mca btl_tcp_if_include eth0*:
  questo parametro indica che l'interfaccia di connessione di rete è la eth0

Questi ultimi due parametri sono stati necessari in quanto, a causa di un bug di OpenMPI all'esecuzione di mpirun nonostante la presenza della sola rete gigabit veniva caricato anche il modulo OpenIB e questo risultava in un deadlock di ORTE.

5.3 I dataset

Per le prove sperimentali sono stati utilizzati due dataset distinti:

- **coaches career (289 oggetti, 7 dimensioni)**: rappresenta le carriere dei coach dell'NBA
- **wine (178 oggetti, 14 dimensioni)**: rappresenta una classificazione dei vini in base a risultati di analisi chimiche

5.4 Dati sperimentali

In questa sezione andremo a raccogliere i dati sperimentali risultanti dalle esecuzioni dell'algoritmo sui due dataset di cui sopra.

Il numero di processi utilizzati per i test è stato **2**,**4**,**8**,**16**,**24**,**32**,**48** e **64**; come si può notare si è sempre scelto un numero pari di processi e questo perchè si è deciso di sfruttare entrambi i core di tutti i nodi.
Il calcolo del tempo di esecuzione è stato fatto internamente all'algoritmo attraverso un timer avviato all'inizio dell'algoritmo e fermato al completamento

dell'algoritmo da parte di tutti i processi.
Per il primo dataset ogni esecuzione è stata ripetuta per 5 volte mentre per il secondo dataset 10 volte, si è poi escluso il valore massimo e minimo e preso la media dei rimanenti valori.

Di seguito le due tabelle con i tempi di esecuzione rilevati, i tempi scartati (massimo e minimo) sono sottolineati, nella prima riga (numero di processi = 1) è indicato il tempo di esecuzione dell'algoritmo nella sua versione sequenziale:

# PROC	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	T MEDIO
1	7	6	5	5	4	**5,33**
2	5	5	4	4	4	**4,33**
4	5	4	4	4	3	**4**
8	2	3	3	3	3	**3**
16	2	2	2	3	3	**2,33**
24	2	2	2	2	2	**2**
32	1	2	1	2	2	**1,67**
48	1	1	1	2	2	**1,33**
64	1	2	2	2	2	**2**

Tabella 5.1: Coaches career dataset: rilevazione tempi di esecuzione

# PROC	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	T6 (s)	T7 (s)	T8 (s)	T9 (s)	T10 (s)	T MEDIO
1	14	15	15	14	15	14	15	14	15	15	**14.5**
2	19	18	18	18	18	18	18	18	18	18	**18**
4	18	17	18	18	18	18	17	17	18	17	**17.63**
8	28	28	28	27	26	28	26	27	27	25	**27.13**
16	18	19	20	19	19	19	19	19	19	20	**19.13**
24	12	12	14	13	13	13	14	13	13	14	**13.13**
32	15	15	12	13	15	14	13	13	14	12	**13.63**
48	16	13	13	15	13	12	15	12	15	11	**13.5**
64	16	15	14	14	14	15	15	15	13	13	**14.38**

Tabella 5.2: Wine dataset: rilevazione tempi di esecuzione

Mostriamo ora le tabelle con lo speedup calcolato a partire dai tempi di esecuzione rilevati.
Ricordiamo che lo speedup per p processi è calcolato dalla seguente formula

$$S_p = \frac{T_1}{T_p}$$

con T_1 tempo di esecuzione dell'algoritmo sequenziale e T_p tempo di esecuzione dell'algoritmo con p processi:

# PROC	TEMPO (s)	SPEEDUP
1	5,33	–
2	4,33	**1,23**
4	4	**1,33**
8	3	**1,78**
16	2,33	**2,29**
24	2	**2,67**
32	1,67	**3,19**
48	1,33	**4,01**
64	2	**2,67**

Tabella 5.3: Coaches career dataset: speedup dell'algoritmo

# PROC	TEMPO (s)	SPEEDUP
1	14.5	–
2	18	**0.81**
4	17.63	**0.82**
8	27.13	**0.53**
16	19.13	**0.76**
24	13.13	**1.10**
32	13.63	**1.06**
48	13.5	**1.07**
64	14.38	**1.00**

Tabella 5.4: Wine dataset: speedup dell'algoritmo

5.5 Riepilogo dei risultati

In questa sezione raccoglieremo i grafici che riepilogano i tempi di esecuzione e lo speedup rilevati al variare del numero di processi per entrambi i dataset utilizzati.

Vediamo dapprima i grafici del tempo di esecuzione e dello speedup per il dataset **coaches career**

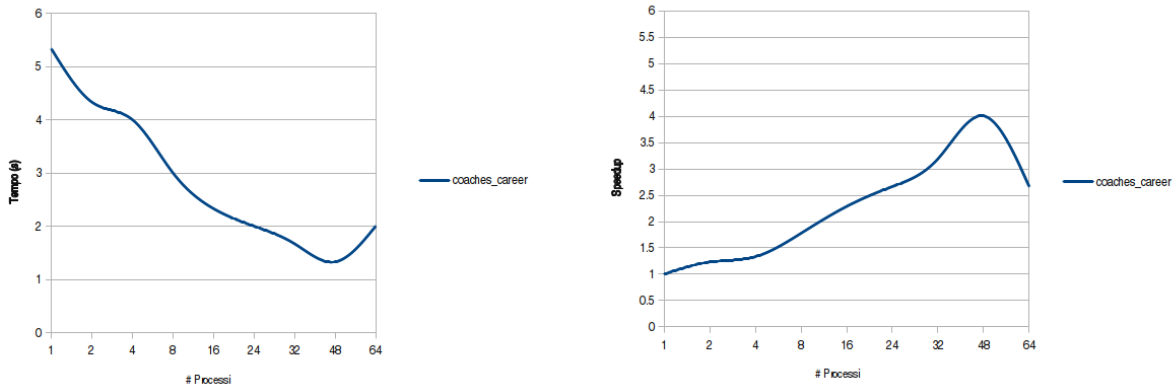


Figura 5.1: Coaches career dataset: grafico tempo e speedup

Come possiamo vedere dai grafici in figura 5.1 nel caso di questo dataset abbiamo uno speedup (grafico a destra) sublineare ma sempre maggiore di 1 e crescente fino ad arrivare ad un massimo di 4.01 per $p = 48$.

Vediamo ora i grafici del tempo di esecuzione e dello speedup per il dataset **wine**

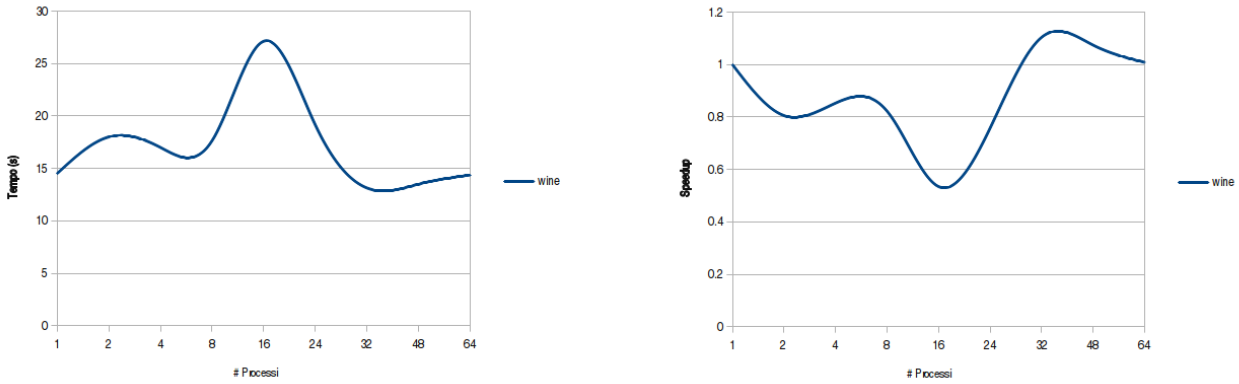


Figura 5.2: Wine dataset: grafico tempo e speedup

In questo caso possiamo notare dai grafici in figura 5.2 come lo speedup (grafico sulla destra) sia per lo più minore di 1 oltre che alquanto irregolare, si ha uno speedup maggiore di 1 solo in un intervallo compreso tra $p = 24$ e $p = 64$.

5.6 Considerazioni

Come si può vedere dai risultati sperimentali presentati nella sezione precedente la parallelizzazione di dINSCY mostra un modesto miglioramento prestazionale nel caso di un dataset "piccolo" (di 7 dimensioni) mentre nel caso di uno significativamente più corposo (di 14 dimensioni) la parallelizzazione non porta nessun grosso vantaggio.
Si è ragionato sul motivo di questi risultati, soprattutto su quelli riguardanti il secondo dataset ovvero **wine** è si è giunti alla conclusione che la principale causa di questo andamento di speedup irregolare possa essere la struttura dati principale utilizzata dall'algoritmo (lo SCY-tree) che per come è definita non è scalabile. Ricordiamo che la strategia di distribuzione prevede che i nodi dello SCY-tree

vengano distribuiti in maniera bilanciata tra i processi, ogni processo eseguirà quindi l'algoritmo sui nodi a lui assegnati.
Per definizione un nodo dello SCY-tree è una coppia (d, i) con d dimensione e i intervallo e rappresenta tutti i punti che nella dimensione d sono posizionati nell'intervallo i.
Da questo si evince che il numero dei nodi che compongono l'albero dipenderà sia dalla dimensionalità del dataset (intesa sia come numero di punti e come numero di attributi) che da come i punti sono posizionati nello spazio: se due punti nella dimensione d ricadono nel medesimo intervallo i di spazio vi sarà un solo nodo che li rappresenta: (d, i); diveramente,se due punti nella dimensione d ricadono rispettivamente nell'intervallo i_1 e i_2 esisteranno due diversi nodi che li rappresentano: (d, i_1) e (d, i_2).
Quindi il carico di lavoro intrinseco in un singolo nodo dello SCY-tree può essere anche molto diverso dal carico di lavoro intrinseco per altri nodi del medesimo SCY-tree; questo si traduce in una possibilità di sbilanciamento del carico di lavoro tra i processi anche se i nodi vengono a loro distribuiti bilanciatamente.
Inoltre un altro fattore che sicuramente può aver influito sui test è da ricercarsi nell'ambiente utilizzato. Ricordiamo che i test sono stati eseguiti sul cluster dipartimentale, le macchine quindi non sono dedicate al calcolo parallelo ma sono utilizzate da tutti gli studenti per i più disparati impieghi; questo si traduce in almeno due conseguenze:

- **risorse disponibili potenzialmente diverse tra macchine**: una macchina può essere impegnata da altri processi (anche computazionalmente pesanti) mentre un altra può essere completamente libera con il risultato di avere uno sbilanciamento di tempi di esecuzione tra processi
- **velocità della rete non garantita**: in certi momenti la rete può risultare congestionata rendendo le comunicazioni tra processi più lente

Conclusioni e sviluppi futuri

L'obiettivo di questo lavoro di tesi è stato quello di implementare una versione parallela dell'algoritmo di subspace clustering INSCY: **dINSCY**.
Non esistendo una implementazione sequenziale di riferimento di INSCY è stato necessario implementare l'algoritmo sequenziale da zero partendo dal solo articolo che lo descriveva dal punto di vista teorico.
A causa di questo l'implementazione sequenziale ha richiesto diversi mesi e la sua realizzazione non è stata esente da problemi rilevanti dovuti al dover tradurre l'articolo, prettamente teorico, in implementazione.
Sempre per la mancanza di un'implementazione di riferimento non è stato possibile eseguire test di tipo qualitativo su dINSCY.
Una volta completata l'implementazione sequenziale si è passato alla progettazione e realizzazione della versione parallela, l'idea base della parallelizzazione è un'idea di per se semplice e intuitiva: distribuire la struttura dati principale dell'algoritmo, lo SCY-tree in maniera bilanciata tra i processi.
Dai risultati delle prove sperimentali si è evinto che questa strategia non ha dato i risultati sperati, infatti se per un dataset di "piccole" dimensioni si è riuscito ad ottenere un piccolo speedup dell'algoritmo con un dataset dimensionalmente più grande non si sono avuti benefici prestazionali dalla parallelizzazione.
Questo risultato è interessante in quanto apre la strada a interessanti sviluppi futuri, potrebbe infatti essere affrontato da due diversi punti di vista:

- la modifica delle strutture dati dell'algoritmo in modo che siano scalabili
- la riprogettazione della parte di parallelizzazione in modo che le unità di lavoro dei singoli processi non siano più i nodi dello SCY-tree

dINSCY non è attualmente ancora stato rilasciato ma si prevede di rilasciarlo a breve sotto licenza open-source GNU GPL.

Appendice A

MPI

MPI (Message Passing Interface) è una specifica di libreria standard per il message-passing.
MPI definisce la sintassi e la semantica di librerie che possono essere usate per scrivere programmi (in C++/Fortran 77) che utilizzano il modello di comunicazione message-passing.
MPI è una specifica di libreria, esistono diverse implementazioni in diversi linguaggi che seguono lo standard MPI; tra le più famose OpenMPI e MPICH (C++/Fortran77).

Lo standard MPI definisce le primitive per la gestione delle topologie virtuali, la sincronizzazione e la comunicazione tra processi. In particolare, le funzionalità di libreria includono primitive di comunicazione sincrone e asincrone di tipo point-to-point (*send/receive*) e collettive (*broadcast/all-to-all*), operazioni di combinazione di risultati parziali di computazione (*gather/reduce*), primitive di sincronizzazione tra processi (*barrier*). Sono inoltre presenti funzionalità di controllo della rete di comunicazione (numero processi attivi, processi vicini nella topologia logica).

A.1 Comunicatore

Il comunicatore MPI definisce "l'universo di comunicazione" ovvero l'insieme dei processi che partecipano nelle comunicazioni point-to-point e collettive.
Esistono tipologie particolari di comunicatori che sono gli **intracomunicatori** ovvero comunicatori che operano all'interno di un singolo gruppo di processi e **intercomunicatori** che operano invece tra gruppi di processi.

I comunicatori sono dinamici, possono essere cioè creati e distrutti durante l'esecuzione del programma.

A.2 Comunicazioni point-to-point

Le comunicazioni point-to-point sono comunicazioni che coinvolgono due singoli processi, uno dei due esegue un'operazione di *send* e l'altro esegue una operazione *receive*.
Nella comunicazione point-to-point sono possibili quattro diverse modalità di invio del messaggio, specificate attraverso la specifica routine *send* usata, tutte le modalità di comunicazione possono essere bloccanti o non bloccanti

- **Sincrona** (MPI_SSend/MPI_ISSend): l'operazione di send può iniziare anche se non è iniziata la corrispondente operazione di receive ma termina solo quando la corrispondente receive sia stata eseguita
- **Buffered** (MPI_BSend/MPI_IBSend): l'operazione di send può iniziare anche se non è iniziata la corrispondente operazione di receive e termina anche se la corrispondente receive non è stata eseguita (il messaggio può essere bufferizzato per garantirne la ricezione).
- **Standard** (MPI_Send/MPI_ISend): l'operazione di send può iniziare anche se non è iniziata la corrispondente operazione di receive, la semantica di completamento può essere sincrona o buffered (a seconda dell'implementazione)
- **Ready**: (MPI_RSend/MPI_IRSend): l'operazione di send può iniziare solo se è stata eseguita la corrispondente receive, il completamento è indipendente dall'esecuzione della corrispondente receive

A.3 Comunicazioni collettive

Le comunicazioni collettive sono comunicazioni che coinvolgono gruppi di processi (tutti appartenenti ad uno stesso comunicatore).
A differenza delle comunicazioni point-to-point le comunicazioni collettive sono sempre bloccanti.
Le comunicazioni collettive si dividono in tre categorie: **sincronizzazione**, **spostamento dati** e **operazioni di riduzione globale**.

A.3.1 Primitive di sincronizzazione

La routine MPI utilizzata per la sincronizzazione tra processi è la *MPI_Barrier (communicator)*, un processo che chiama questa routine su un comunicatore rimane bloccato fino a quando tutti i processi del comunicatore non abbiano a loro volta chiamato questa routine.

A.3.2 Primitive di spostamento dati

Le primitive per lo spostamento collettivo dei dati sono: *MPI_Broadcast*, *MPI_Scatter*, *MPI_Gather* e *MPI_AllToAll*.
La *MPI_Broadcast* invia un insieme di dati da un processore nel comunicatore (detto radice) e tutti gli altri processori nel comunicatore, tutti i processi coinvolti devono conoscere il processore radice.
La *MPI_Scatter* prende in input un array di elementi, li divide in n parti e invia ogni parte i al processo con rank i.
La *MPI_Gather* esegue l'operazione inversa della *MPI_Scatter*, quando chiamata da ogni processo questo invia il contenuto del suo buffer al processo radice (che deve essere conosciuto da tutti i processi coinvolti nell'operazione) il quale combina le parti ricevute a seconda del rank del mittente.
Una variante della *MPI_Gather* è la *MPI_AllGather* in cui non solo il processo radice riceve il risultato ma tutti i processi coinvolti.
La *MPI_AllToAll* lavora similmente alla *MPI_AllGather* con la differenza che ogni processo invia dati diversi a processi diversi.

A.3.3 Primitive di riduzione globale

La primitiva di riduzione globale di MPI prende il nome di *MPI_Reduce*. Quando questa routine viene chiamata da ogni processo in un comunicatore vengono combinati i dati presenti nei buffer di invio di ogni processo usando l'operazione di riduzione specificata e i risultati vengono memorizzati nel buffer di ricezione del processo radice.
Le operazioni di riduzione possono essere aritmetiche (+,-,*,/), logiche (and, or, and bit-a-bit, or bit-a-bit), confronto (min, max).

Bibliografia

[1] Rakesh Agrawal, Johannes Gehrke, Dimitros Gunopulos, Prabhakar Raghavan. Automatic subspace clustering of high dimnesional data for data mining applications.

[2] I.Assent R.Krieger E.Muller T.Seidl. Inscy: Indexing subspace clusters with in process removal of redundancy. *8th IEEE International Conference on Data Mining*, 2008.

[3] http://www.infovis-wiki.net.

[4] Wikipedia: Dbscan
http://it.wikipedia.org/wiki/dbscan.

[5] Wikipedia: Data mining
http://en.wikipedia.org/wiki/data_mining.

[6] M.Brescia. Review on clustering in data mining.

[7] M.Ester H.P.Kriegel J.Sander X.Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. 1996.

[8] H.Liu L.Parsons, E.Haque. Subspace clustering for high dimensional data: A review. *Sigkdd Explorations*, 2004.

[9] I. T. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[10] Singular value decomposition
http://en.wikipedia.org/wiki/singular_value_decomposition.

[11] Sanjay Goil,Harsha Nagesh, Alok Choudhary. Mafia: Efficient and scalable subspace clustering for very large data sets.

[12] K.Kailing H.P. Kriegel P. Kroger. Density-connected subspace clustering for high-dimensional data. *4th Siam International Conference on Data Mining (pp.246-257)*, 2004.

[13] I.Assent R.Krieger E.Muller T.Seidl. Edsc: Efficient density based subspace clustering. *CIKM*, 2008.

[14] Renato Agati. Calcolo parallelo.

Ringraziamenti

Il primo ringraziamento va alla mia ragazza Giulia che mi ha regalato e mi sta regalando i migliori anni della mia vita e che in particolare in questi ultimi mesi mi ha supportato (e sopportato) tanto. Il secondo ringraziamento va a tutta la mia famiglia: ai miei genitori Marcello e Liliana e mia sorella Francesca che mi sono sempre vicini nel bene e nel male e che nei momenti più difficili sanno sempre consigliarmi al meglio ma anche ai miei zii Leana e Vittoriano e ai miei cugini Mauro e Simone che anche se non vedo spessissimo sento sempre vicini.
Il terzo ringraziamento va ai miei amici, in particolare a tutti coloro che mi hanno accompagnato nel mio percorso di maturazione all'università, quelli di vecchia data Enrico, Federico, Davide, Dario, Matteo ... i più "recenti" Luca, Marco, Massimo, Alessio, Jacopo, i ragazzi del "delirio" e tutti gli altri che, anche se non nomino esplicitamente, sanno benissimo di avere sempre un posto speciale nella mia testa e nel mio cuore.
Ringrazio inoltre i miei colleghi (anzi per lo più colleghe) di YU che hanno costituito in questi mesi una bella valvola di sfogo per uscire ogni tanto dallo "stress da tesi"

Un ringraziamento speciale infine va al relatore dottor Moreno Marzolla, al correlatore dottor Matteo Magnani e alla dottoressa Ira Assent che con la loro grande disponibilità e pazienza hanno dato un contributo fondamentale alla realizzazione di questo lavoro di tesi.