ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA
Dipartimento di Informatica – Scienza e Ingegneria

TESI DI LAUREA

in
Processi e tecniche di Data Mining M

ALGORITMI PER LA SCOPERTA DI OUTLIER IN SISTEMI DI CALCOLO AD ALTISSIME PRESTAZIONI

CANDIDATO
Matteo Rasenti

RELATORE
Chiar.mo Prof. Ing. Claudio Sartori

CORRELATORI
Prof. Ing. Stefano Lodi
Ing. Stefano Basta

Anno Accademico 2012/2013

Sessione III

Indice

1 Introduzione 2
2 Outlier Detection Problem 4
2.1 Tecniche supervisionate e non supervisionate 5
2.2 Principali approcci di outlier detection 6
2.3 Il problema dell'alta dimensionalità 9
2.4 Outlier detection in dataset distribuiti 10
3 L'algoritmo DistributedSolvingSet 11
3.1 Concetti principali 11
3.2 L'algoritmo SolvingSet 12
3.3 L'algoritmo DistributedSolvingSet 13
3.3.1 Parametri di ingresso 14
3.3.2 Passi dell'algoritmo 14
3.3.3 Procedure eseguite sui nodi locali 15
3.4 Correttezza dell'algoritmo 17
3.5 Costo 18
3.5.1 Costo temporale 18
3.5.2 Costo di trasmissione 19
4 L'algoritmo LazyDistributedSolvingSet 21
4.1 Passi dell'algoritmo 21
4.2 Procedure eseguite sui nodi locali 24
5 MPI, Message Passing Interface 26
5.1 Breve storia di MPI 26
5.2 Concetti chiave 28
5.2.1 Communicator e rank 28
5.2.2 Comunicazione 28
5.3 Inizializzazione dell'environment MPI 29
5.4 Comunicazione point-to-point 30
5.5 Comunicazioni collettive 31
5.6 Tipi di dato 33
6 Implementazione dell'algoritmo 35
6.1 Interazione master e slave 35
6.2 Punti e candidati 36
6.3 Heap 37
6.4 Organizzazione dei namespace 38
6.5 Le librerie Boost 39
6.5.1 Build delle librerie 40
6.5.2 Utilizzo 41
7 Risultati sperimentali 44
8 Conclusioni 54
9 Bibliografia 57

1 Introduzione

Secondo la definizione di Hawkins, un outlier è un'osservazione così diversa dalle altre da poter causare il sospetto che sia stata generata tramite qualche altro meccanismo. Questa definizione è basata su considerazioni statistiche e assume che gli oggetti normali seguano un comune "meccanismo di generazione", come un determinato processo statistico. Gli oggetti che si discostano da questo meccanismo vengono considerati outlier. Generalizzando questa definizione possiamo definire gli outlier come dei pattern che non rispecchiano il normale comportamento atteso.

Gli outlier possono essere generati da diverse cause quali guasti, errori di misura, errori umani e deviazioni naturali. Queste osservazioni non rappresentano sempre degli errori o delle informazioni alterate durante la fase di campionamento o di preprocessing.

In alcuni casi gli outlier vengono identificati per poterli escludere dall'analisi, perché le loro caratteristiche possono alterare in modo significativo i valori medi che verranno utilizzati a fini statistici. In altri campi sono invece gli outlier ad avere importanza. Ad esempio, nel campo della frode fiscale un outlier può rappresentare un'attività fraudolenta, come un acquisto tramite una carta di credito denunciata precedentemente come smarrita, o una serie di pagamenti che possono caratterizzare un abuso. Nel campo medico, invece, gli outlier possono essere dei sintomi poco comuni o delle caratteristiche inaspettate risultate da una certa analisi.

Il problema della scoperta di outlier è quindi applicato in diversi ambienti, come i già citati frode fiscale e il campo medico, ma anche nei campi dell'analisi di robustezza delle reti, analisi sulla popolazione, predizioni meteorologiche e analisi di mercato. Questi ambienti hanno due cose in comune: richiedono un'analisi su una grande quantità di dati e necessitano di avere i risultati in brevi termini di tempo.

La grande quantità di informazioni da processare rende una normale workstation inadatta a questo lavoro. Ecco quindi che l'analisi viene eseguita in ambienti HPC, High Performance Computing, che realizzano dei sistemi di elaborazione in grado di fornire prestazioni molto elevate nell'ordine dei petaFLOPS[1]. Per sfruttare al meglio queste incredibili potenze di calcolo, è necessario utilizzare algoritmi di analisi che non adottano più un approccio sequenziale, ma parallelo. L'idea è quella di individuare e sfruttare le proprietà locali del

1 Un petaFLOPS equivale a 10^{15} operazioni in virgola mobile eseguite in un secondo dalla CPU

problema in modo da partizionare il calcolo tra i nodi del sistema e decrementare così i tempi di esecuzione.

In questa tesi verrà analizzato l'algoritmo parallelo/distribuito *DistributedSolvingSet* proposto da Fabrizio Angiulli, Stefano Basta, Stefano Lodi e Claudio Sartori, e la sua variante, il *LazyDistributedSolvingSet*. Entrambi gli algoritmi sono basati sul concetto di solving set esteso al caso dei dataset distribuiti. Il solving set S è un sottoinsieme di elementi del dataset originale D che include un numero sufficiente di oggetti tali da permettere di considerare solamente le distanze tra le coppie in $S \times D$. È quindi possibile vedere il solving set come una rappresentazione compressa del dataset originale, e può essere usato per sapere se un certo oggetto è un outlier o meno confrontandolo solamente con gli elementi appartenenti ad esso.

Lo scopo di questa tesi è implementare l'algoritmo LazyDistributedSolvingSet, eseguire dei test e valutare le performance. I test sono stati eseguiti su *FERMI*, un supercomputer IBM Blue Gene/Q, presente presso il centro di calcolo del CINECA.

Il prossimo capitolo fornirà una panoramica al problema della scoperta degli outlier, mentre nei capitolo 3 e 4 vedremo in dettaglio l'algoritmo ODP *DistributedSolvingSet* e la sua variante, il *LazyDistributedSolvingSet*.

I capitoli successivi sono più tecnici: il capitolo 5 è un'introduzione a MPI, Message Passing Interface, l'interfaccia di scambio di messaggi standard de-facto usata nella comunicazione tra processi in ambienti HPC; il capitolo 6 parlerà dell'implementazione dell'algoritmo e descriverà i dettagli principali del software, che potranno essere utili come punto di inizio per chi, dopo di me, dovrà lavorare sul progetto per estenderlo.

Nel capitolo 7 verranno invece mostrati i test eseguiti, i risultati ottenuti e le osservazioni riscontrate. Infine, nel capitolo 8 verranno tratte le conclusioni e verranno suggeriti degli spunti per eventuali sviluppi futuri.

2 Outlier Detection Problem

Un outlier, chiamato anche *anomalia*, è un particolare dato che presenta delle caratteristiche molto diverse da quelle degli altri dati presenti del dataset. In generale un outlier non segue il comportamento generale o il modello dei dati e in molti casi contiene informazioni importanti riguardo caratteristiche non comuni dell'ambiente in esame. Il riconoscimento di queste caratteristiche inusuali è molto importante in ambienti quali:

- IDS, Intrusion Detection Systems: monitorando le attività correnti su un host o sulla rete è possibile individuare attività inusuali e etichettarle come attività maligne o accessi non autorizzati.
- Frodi finanziarie: utilizzi anomali delle carte di credito possono indicare un potenziale furto.
- Diagnosi mediche: pattern inusuali ottenuti da diverse analisi possono essere associati a specifiche malattie.

Le azioni che seguono la scoperta degli outlier dipendono dall'area di applicazione. Se l'outlier identifica un errore tipografico, allora quell'errore sarà corretto. Se un outlier rappresenta un errore di misura, allora potrà essere omesso. In uno studio sulle caratteristiche di una popolazione potremmo trovare qualche persona molto alta; in questo caso l'anomalia potrebbe essere del tutto naturale, quindi ci si potrebbe accertare che non sia un errore e in caso positivo includerla nello studio.

In molte applicazioni, i dati hanno un modello di comportamento “normale” e gli outlier vengono riconosciuti perché non rientrano in tale modello. In molti casi invece gli outlier possono essere individuati solo come sequenza di dati. Qui torniamo all'esempio già citato della frode fiscale, in cui ciascun dato rappresenta un'azione, e una particolare sequenza di azioni può identificare una frode. In queste situazioni le anomalie sono chiamate anche *anomalie collettive*, perché possono essere identificate solo come un insieme o come una sequenza di dati apparentemente normali se presi singolarmente.

Il risultato dell'analisi può essere di due tipi: il più semplice prevede un'uscita binaria “sì/no” che indica se un certo dato è un outlier o meno; in alternativa può venire restituito uno peso, o

“score”, che indica quanto un certo dato è un outlier. Più il peso è alto e più il dato ha caratteristiche anomali. Il peso può essere usato anche per determinare il ranking dei top-n outlier.

La maggior parte degli algoritmi individua gli outlier assegnandogli proprio un peso. Il peso può essere calcolato tramite considerazioni sulla sparsità della regione, considerazioni sulle distanze dai vicini o il match con una certa distribuzione di dati.

Anche se non avviene sempre in modo esplicito, ogni approccio di outlier detection traccia un modello a cui sottostanno i dati definiti “normali”, e in molti casi il modello è definito dall'algoritmo. Ad esempio, i metodi basati sulle distanze nearest-neighbor modellano l'outlier tendency di un punto in termini della distanza del punto dai suoi primi k vicini. In questo modo si assume che gli outlier siano localizzati a grandi distanze dalla maggior parte dei dati. La scelta del modello è importante perché diversi modelli possono condurre diversi risultati. Ad esempio, un modello rappresentato come una gaussiana può produrre risultati scadenti se i dati non fanno veramente riferimento a quel modello.

Per l'analisi, i dati possono essere rappresentati in maniera diversa. Spesso vengono rappresentati come dei punti su uno spazio multidimensionale, in cui ciascuna dimensione rappresenta una certa proprietà del dato. In molti casi i dati hanno così tante proprietà da prendere in esame che per semplificare l'analisi vengono mappati in uno spazio con un numero di dimensioni inferiore a quello originale.

2.1 Tecniche supervisionate e non supervisionate

In generale possiamo dividere le tecniche di outlier detection in due categorie: quelle che hanno a disposizione un training-set dei dati e quelle che invece non lo hanno. Nel primo caso rientrano sia le tecniche supervisionate che quelle semi-supervisionate, e sono basate sull'idea di usare il training-set per costruire un modello di predizione per classificare un certo elemento come un outlier o meno. Sono quindi basate su classificatori come reti bayesiane o reti neurali.

Nelle tecniche supervisionate il training set è completo, ovvero contiene esempi di dati normali e di dati anomali. Le tecniche semi-supervisionate dispongono invece di un training-set contenente solo esempi di dati normali. Il modello di predizione di queste ultime sarà ovviamente meno accurato di quello delle prime, dato che è costruito sulla base di un training-

set più povero. Tuttavia, le tecniche semi-supervisionate sono più applicabili perché non sempre si hanno a disposizione dei campioni di outlier.

In uno scenario non supervisionato non troviamo invece nessuna guida e l'algoritmo deve individuare gli outliers solamente sulla base dei dati a disposizione e sull'assunzione che le istanze di dati normali siano nettamente superiori in numero alle istanze delle anomalie. Questi sono gli approcci più applicabili proprio perché l'unica cosa che richiedono per poter essere eseguiti sono i dati da analizzare.

Le tecniche supervisionate sono tipicamente più efficienti di quelle non supervisionate, perché la conoscenza posseduta nei training-set viene usata per affinare il processo di ricerca. In molti casi però, i pattern che rappresentano il comportamento normale sono in continua evoluzione e una rappresentazione attuale di comportamento normale può non essere valida nel futuro. Inoltre non sempre si hanno a disposizione i training-set di esempio.

2.2 Principali approcci di outlier detection

Approcci statistici

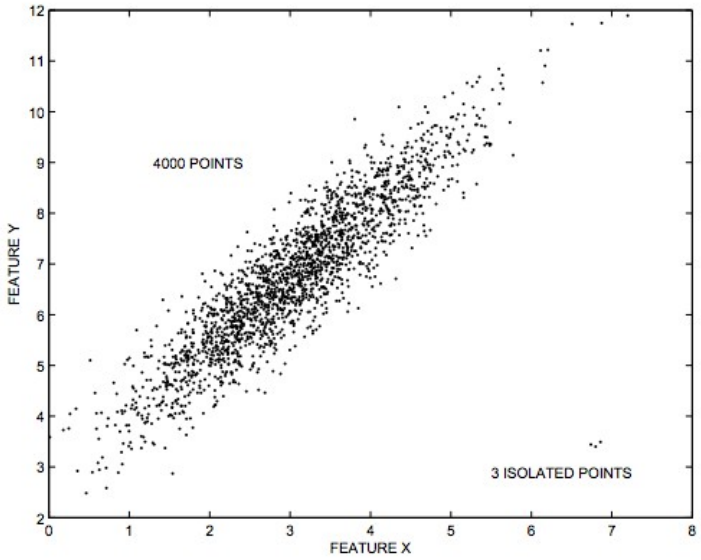
Gli approcci statistici fanno l'assunzione che i dati siano stati generati da una specifica distribuzione. Secondo questa assunzione gli outlier saranno quindi quei punti che hanno poca probabilità di essere stati generati secondo tale distribuzione. La loro efficacia dipende ovviamente da quanto in realtà i dati seguono il modello che descrive la loro distribuzione, quindi la sua scelta è molto importante.

Approcci basati sulla prossimità

L'idea comune a questo genere di approcci è quella di modellare gli outlier come punti isolati dai dati normali, senza fare alcuna assunzione sulla loro distribuzione. Gli approcci basati sulla prossimità si dividono in tre tipi: quelli basati sui nearest neighbors, i vicini più prossimi; quelli basati sulla densità; quelli basati sul clustering.

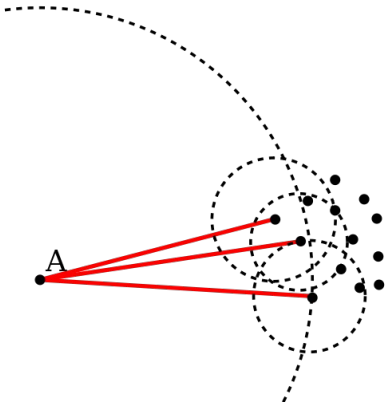
Negli approcci nearest neighbor, tipicamente viene determinata la distanza di un punto dai suoi primi k vicini. La scelta del valore k è molto importante ai fini della soluzione. Con un k troppo piccolo, piccoli gruppi di punti, vicini tra loro ma lontani dal resto dei dati possono non essere etichettati come outlier. Nella figura successiva, i tre punti raggruppati in basso a destra sono chiaramente degli outlier, ma se viene usato un k troppo piccolo, ad esempio k=2,

verranno riconosciuti come punti normali proprio perché sono molto vicini tra loro. L'approccio naif, che consiste nel calcolare, per ciascun punto del dataset, il suo peso come funzione delle k distanze dai suoi primi k vicini, ha una complessità di $O(N^2)$, dove N è il numero di punti del dataset.

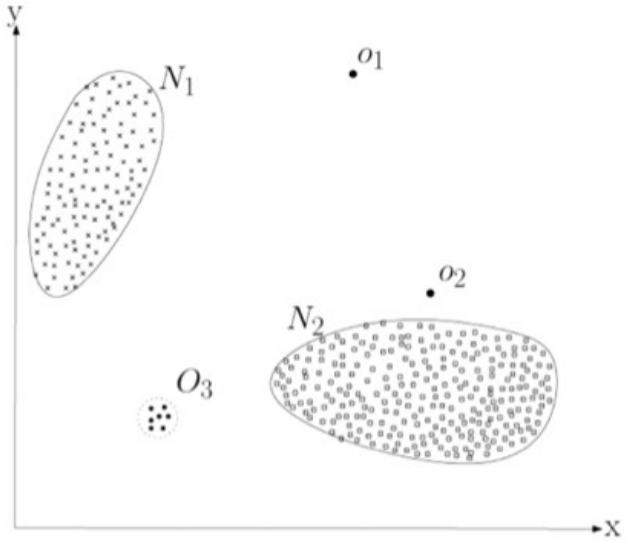


Gli approcci basati sulla densità calcolano, per ogni punto, la densità del suo vicinato. Gli outlier saranno quei punti che risiedono in un vicinato a bassa densità. Anche qui la densità è definita in base alla distanza di un punto dai suoi vicini, ma in generale abbiamo diverse interpretazioni.

Nel metodo *LOF*, Local Outlier Factor, lo score di un punto è definito come il rapporto tra la densità media locale dei *k*-NN di *p* e la densità locale del punto stesso. Per il calcolo della densità locale di *p*, viene trovato il raggio dell'ipersfera centrata nel punto contenente i *k*-NN e posto il valore della densità locale pari al volume della ipersfera diviso *k*. Un punto normale apparterrà a una regione densa e il suo *LOF* sarà prossimo a 1, mentre un outlier avrà un *LOF*>> 1. La figura sottostante mostra il concetto del calcolo della densità locale, con *k*=3:



Gli approcci basati sul clustering partizionano i dati in cluster di dimensioni e densità variabile. Sono basati sull'assunzione che i dati normali appartengono a cluster di grandi dimensione e alta densità, mentre gli outlier appartengono a cluster piccoli di bassa densità, o anche a nessun cluster. Gli outlier saranno quei punti presenti nei cluster di dimensione o densità inferiore un valore di soglia prestabilito. Tipicamente. in un approccio di questo tipo, il primo passo è quello di usare un algoritmo di clustering per determinare le regioni più dense. Successivamente si etichettano come candidati i punti appartenenti a cluster piccoli e si calcola la distanza tra questi e i cluster non candidati: se i punti candidati sono lontani da tutti i punti non candidati, allora sono dei veri outlier. Nella figura sottostante i punti appartenenti al cluster O_3 verranno identificati come outlier, e lo stesso vale per i punti O_1 e O_2. I cluster N_1 e N_2 vengono invece identificati come classi normali.



Approcci grafici

Basati sull'osservazione degli oggetti come punti mappati in uno spazio multidimensionale, è un tipo di analisi svolta dall'operatore umano, e per questo motivo è soggettiva. Inoltre, la complessità di questo tipo di analisi cresce notevolmente all'aumentare della dimensionalità dello spazio da osservare. Una possibile tecnica è quella di definire il *convex hull*, un involucro che racchiude nello spazio tutti i punti normali, e considerare come outlier tutti i punti al di fuori di esso. Tuttavia definire una regione di questo tipo, senza che includa anche qualche outlier, può essere difficile.

2.3 Il problema dell'alta dimensionalità

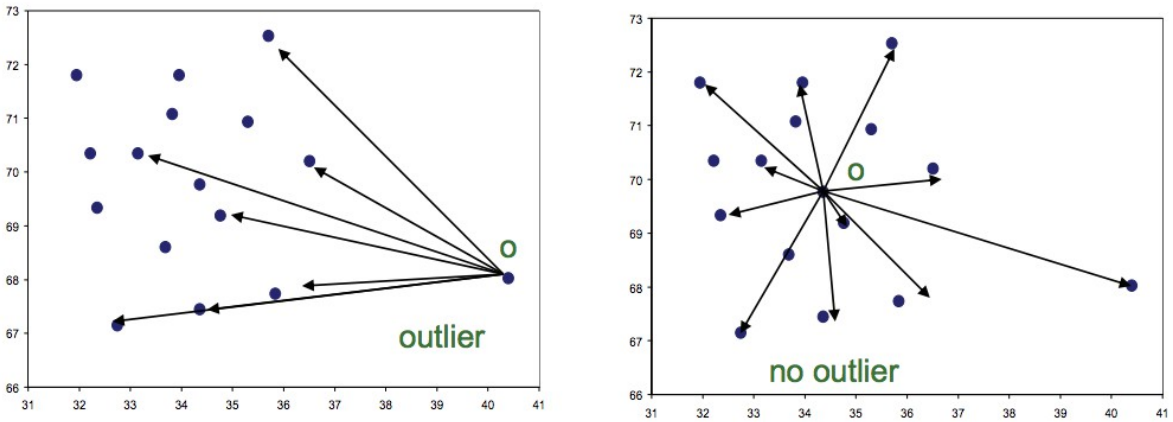
Per dataset con elevata dimensionalità, i dati sono sparsi e le definizioni di prossimità, distanza, densità e vicinato diventano poco significative, tanto che alcune tecniche di analisi perdono la loro efficacia. Questo problema è anche noto come la *maledizione della dimensionalità*.

In questi casi si ricorre spesso a tecniche basate sulla proiezione dei punti su spazi a dimensionalità minori. Un punto viene definito outlier se in qualche proiezione si trova in una porzione di spazio con densità di punti molto bassa. In questo modo è anche più facile capire quali sono le proprietà che rendono una particolare informazione un outlier.

Il problema diventa quindi determinare quali regioni delle proiezioni hanno una bassa densità e identificare come outlier gli oggetti proiettati su di esse. L'individuazione di regioni a bassa densità avviene comparando la densità effettiva della regione con la densità che dovrebbe avere la regione assumendo una distribuzione uniforme dei punti nello spazio.

Una soluzione alternativa ricorre invece all'utilizzo di funzioni distanza più robuste. Un esempio è la funzione Angle-Based Outlier Degree, basata sull'osservazione che in uno spazio multidimensionale gli angoli sono più stabili delle distanze. Di conseguenza un oggetto è un outlier se la maggior parte degli altri oggetti sono localizzati in direzioni simili. Al contrario un oggetto non è un outlier se molti altri oggetti sono localizzati in direzioni diverse.

La figura successiva mostra un esempio applicato, per semplicità, in uno spazio a due dimensioni.

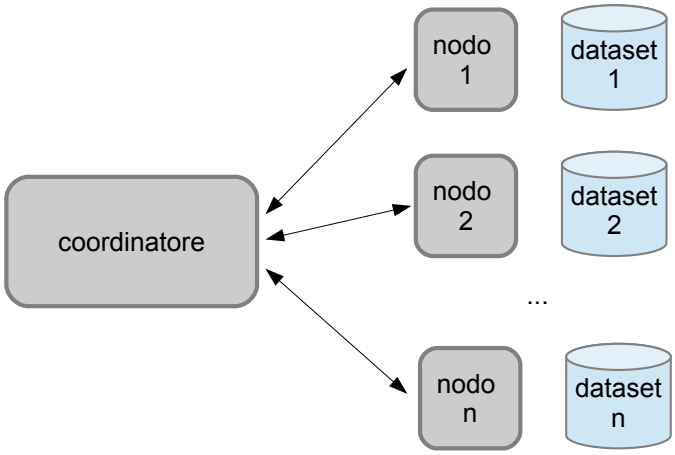


2.4 Outlier detection in dataset distribuiti

Nel data mining le informazioni da analizzare provengono spesso da fonti diverse e in generale possono essere distribuite su un gruppo di nodi indipendenti, ciascuno dotato della sua potenza di calcolo, dispositivi di memorizzazione, e di una connessione alla rete.

Una soluzione al problema, immediata come ragionamento ma non molto efficiente, è sicuramente quella di centralizzare il dataset su un unico nodo e eseguire l'analisi mediante un approccio centralizzato. Purtroppo la fattibilità di questa soluzione è legata fortemente alla dimensione del dataset globale, e le performance calano al suo aumentare. Inoltre sono da considerare anche i tempi necessari al trasferimento dei dataset distribuiti.

Un approccio più furbo è invece quello di sfruttare in parallelo la potenza di calcolo di ciascun nodo che ospita una parte del dataset globale. Ogni nodo eseguirà un algoritmo e invierà i risultati parziali a un nodo coordinatore, il quale li elaborerà al fine di produrre il risultato finale. In questo modo si riducono i tempi di esecuzione e si riduce anche il carico della rete, dato che vengono trasferiti solo alcuni risultati parziali invece che l'intero dataset.



L'algoritmo ODP *DistributedSolvingSet*, che verrà presentato nel prossimo capitolo, è proprio basato su questi principi. Il suo scopo è quello di sfruttare le proprietà locali del problema in modo da dividere il calcolo tra i nodi disponibili e risolvere più velocemente il problema.

3 L'algoritmo DistributedSolvingSet

L'algoritmo *DistributedSolvingSet* è un algoritmo di outlier detection non supervisionato e distance-based, orientato al calcolo parallelo, mirato a trovare una soluzione nel minor tempo possibile lavorando su dataset distribuiti e di grandi dimensioni.

3.1 Concetti principali

Solving Set

L'algoritmo è basato sul concetto di solving set S, un sottoinsieme di elementi del dataset originale D che include un numero sufficiente di oggetti tali da permettere di considerare solamente le distanze tra le coppie in $S \times D$. È possibile vedere il solving set come una rappresentazione compressa del dataset originale, e lo si può usare per sapere se un certo oggetto è un outlier o meno confrontandolo solamente con gli elementi appartenenti a questo insieme.

Il solving set contiene almeno i top-n outlier, quindi risolvere il problema del calcolo del solving set risolve anche il problema del calcolo dei top-n outlier.

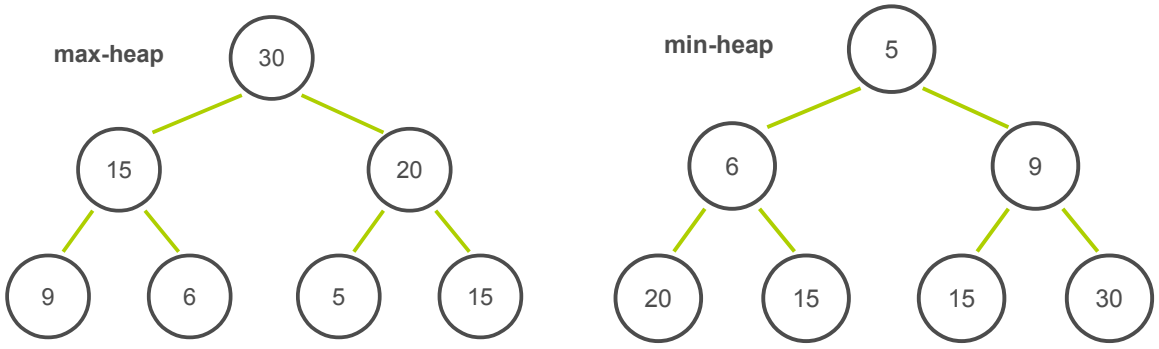
Distance-based

Un oggetto è definito un outlier o meno a seconda del suo peso, che è funzione delle distanze dell'oggetto dai suoi k oggetti più vicini. Secondo questo concetto, gli outlier sono gli n oggetti avente peso più grande.

Heap

L'algoritmo usa delle strutture heap binarie per mantenere ordinate diverse informazioni. Una heap è una struttura dati organizzata come un albero binario in cui:

- nel caso di una min-heap, fissata una certa funzione di comparazione, un qualunque elemento padre ha sempre un valore minore o al limite uguale al valore dei suoi figli. L'elemento radice sarà quindi l'elemento con valore più grande;
- nel caso di una max-heap, l'elemento padre ha sempre un valore maggiore o uguale al valore dei suoi figli. L'elemento radice sarà l'elemento con valore più piccolo.



Parametri di ingresso

- *n*: numero di top outlier da trovare;
- *k*: numero di oggetti vicini da considerare per il calcolo del peso di un oggetto;
- *m*: numero di oggetti che verranno aggiunti al solving set a ogni iterazione;

3.2 L'algoritmo SolvingSet

L'algoritmo DistributedSolvingSet è una estensione dell'algoritmo sequenziale *SolvingSet.* Per semplicità è bene quindi cominciare dalla descrizione di questo algoritmo.

Nell'algoritmo *SolvingSet*, ciascun oggetto del dataset contiene una max-heap delle distanze dai suoi primi *k* vicini incontrati finora, e il suo peso attuale è funzione di queste distanze.

Il peso degli oggetti è calcolato lungo diverse iterazioni. A ogni iterazione ciascun punto del dataset viene confrontato solo con gli oggetti di un piccolo sottoinsieme del dataset stesso. Questo sottoinsieme si chiama *insieme dei candidati,* e gli oggetti che fanno parte dell'insieme dei candidati usato nella prima iterazione sono scelti in maniera casuale.

Ogni confronto eseguito implica un possibile aggiornamento delle heap delle distanze dei due punti coinvolti. L'aggiornamento avviene solo se la heap non è piena (ovvero contiene un numero di distanze < *k*), o se la nuova distanza calcolata è inferiore alla distanza dall'attuale *k*-esimo vicino. In quest'ultimo caso il peso del punto in considerazione diminuirà.

Dato che gli oggetti candidati vengono confrontati con tutti i punti del dataset, il loro peso reale è noto. Per gli altri oggetti, invece, il loro peso attuale rappresenta un upper bound del

loro peso reale. Se questo valore è inferiore al peso dell'*n*-esimo candidato con peso più grande, allora l'oggetto non potrà mai far parte dell'insieme dei top-*n* outlier e verrà etichettato come *non attivo*.

Alla fine di un'iterazione vengono selezionati, tra gli oggetti rimasti attivi, quelli aventi il massimo peso corrente e verranno utilizzati per formare il nuovo insieme di candidati per l'iterazione successiva.

Iterazione dopo iterazione, i pesi degli oggetti non candidati sono sempre più accurati e il numero degli oggetti dichiarati non attivi aumenta. L'algoritmo termina quando non si hanno più oggetti dichiarati attivi, e il solving set sarà dato dall'unione dei vari insiemi dei candidati usati a ogni iterazione. I top-*n* outlier saranno così gli *n* oggetti con peso più grande all'interno del solving set.

3.3 L'algoritmo DistributedSolvingSet

L'algoritmo DistributedSolvingSet è stato pensato per essere eseguito in uno scenario distribuito dotato di un nodo supervisione N_0, e con un dataset partizionato lungo una serie di nodi locali N_i ($i = 1, \ldots l$). Come il suo predecessore, anche questo calcola il solving set e risolve il problema della scoperta dei top-*n* outlier.

L'algoritmo alterna due fasi: una fase di computazione, eseguita sui nodi locali, e una fase di sincronizzazione dei risultati parziali restituiti da ciascun nodo, eseguita sul nodo coordinatore.

Durante la fase di computazione, ogni nodo locale riceve dal coordinatore l'insieme corrente dei candidati e il lower bound, ovvero il peso dell'attuale *n*-esimo top outlier. Ciascun nodo confronta gli oggetti ricevuti con tutti gli oggetti presenti nel suo dataset locale, e restituisce al coordinatore un nuovo set di candidati locali per l'iterazione successiva e la lista dei *k* vicini più prossimi locali rispetto ai candidati ricevuti. Viene restituito anche il numero di oggetti locali attivi, ovvero gli oggetti con peso non inferiore al lower bound ricevuto.

Durante la fase di sincronizzazione, il coordinatore utilizza i dati ricevuti per calcolare il peso esatto di ogni candidato, aggiornare l'insieme corrente dei top-*n* outlier e generare un nuovo insieme di candidati per l'iterazione successiva.

3.3.1 Parametri di ingresso

L'algoritmo riceve in ingresso il numero l di nodi locali, la dimensione d_i dei dataset locali, il k numero di vicini da considerare per il calcolo del peso di un punto, il numero n di top outlier da trovare, e un parametro $m \geq k$ che specifica il numero di oggetti da aggiungere al solving set a ogni iterazione.

3.3.2 Passi dell'algoritmo

Inizialmente, il solving set e il solution set sono entrambi vuoti e il lower bound è fissato a 0, mentre il numero totale di nodi attivi è inizializzato alla dimensione del dataset globale.

Il coordinatore effettua la chiamata a una procedura di inizializzazione, *NodeInit()*, che verrà eseguita su tutti i nodi locali. Ogni nodo locale N_i inizializzerà le proprie strutture e sceglierà casualmente m_i oggetti dal suo dataset, che invierà al coordinatore.

Il coordinatore crea l'insieme dei candidati globali raggruppando gli oggetti ricevuti dai nodi locali e comincia una fase iterativa. Alla fine di ciascuna fase viene eletto un nuovo insieme di candidati per la prossima iterazione. Se questo insieme è vuoto l'algoritmo termina.

In ogni iterazione, per prima cosa il coordinatore aggiunge l'insieme corrente dei candidati al solving set, poi chiede ai nodi locali di eseguire la procedura di computazione, *NodeComp()*. Con questa chiamata, i nodi locali ricevono il lower bound attuale, l'insieme dei candidati correnti e il numero totale di oggetti attivi. Come risposta, il coordinatore si aspetta da ciascun nodo locale:

- una collezione *LNNC* di m elementi, ciascuna contenente le k distanze dei vicini più prossimi rispetto a ciascun candidato globale;
- una max-heap *LC*, contenente i punti locali che faranno parte dell'insieme dei candidati nell'iterazione successiva;
- il numero di punti locali rimasti attivi.

Il coordinatore utilizza le informazioni ricevute per aggiornare il numero globale di nodi attivi e per calcolare per ogni candidato il suo vero peso sulla base delle distanze ricevute. Se il candidato ha un peso maggiore dell'attuale lower bound, allora viene aggiunto alla heap degli attuali top-n e il lower bound viene quindi aggiornato. Il nuovo set di candidati viene costruito come l'unione dei candidati ricevuti da ciascun nodo locale e, se questo insieme non è vuoto, comincia una nuova iterazione.

Pseudo codice

```
DistributedSolvingSet(l, [d1, … , dl], dist, n, k, m) {
   DSS = Ø;
   OUT = Ø;
   d =  i di;

   for each node Ni in N
      NodeInit(mi, Ci);

   C =  i Ci;
   act = d;
   minOUT = 0;

   while (C != Ø) {
      DSS = DSS    C;
      for each node Ni in N
         NodeComp(minOUT, C, act, LNNCi, LCi, acti);

      act =  i acti;

      for each (q in C) {
         NNC[q] = get_k_NNC( i LNNCi[q]);
         UpdateMax(OUT, (q, Sum(NNC[q])));
      }

      minOUT = Min(OUT);
      C = Ø;

      for each (p in  i LCi) {
         C = C    {p};
      }
   }
}
```

3.3.3 Procedure eseguite sui nodi locali

NodeInit

Procedura di inizializzazione, eseguita una sola volta. Riceve in ingresso il numero intero m_i e restituisce un insieme di m_i oggetti locali scelti in maniera casuale. Questo insieme viene conservato anche sul nodo locale. Viene inoltre inizializzata la variabile act_i, che rappresenta il numero di oggetti attivi nel dataset locale.

```
NodeInit_i(mi, Ci){
   Ci = RandomSelect(Di, mi);
   acti = |Di|;
   store(acti, Ci);
}
```

NodeComp

Procedura di computazione, viene eseguita a ogni iterazione. Riceve in input il lower bound *minOUT*, ovvero il peso dell'attuale n-esimo outlier, l'insieme dei candidati *C* e il numero totale di oggetti attivi *act*.

La procedura comincia con il caricamento dei candidati locali scelti nella NodeComp precedente (o nella NodeInit, se è la prima chiamata a NodeComp), e del numero di oggetti locali attivi. Si rimuovono i candidati locali dal dataset locale e si aggiorna il conto degli oggetti locali attivi. Dopodiché si inizializza la heap $LNNC_i$ con il contenuto dei *NN* per ogni candidato locale. Gli heap per i punti candidati non locali vengono inizializzati come vuoti, perché solo il nodo locale che li ha generati si deve preoccupare di memorizzare i *k* vicini più prossimi di tali punti rispetto ai precedenti set di candidati.

A questo punto ciascun candidato locale viene confrontato con tutti gli oggetti locali. Questo procedimento è diviso in tre parti:

1. Ciascun candidato locale viene confrontato con ogni altro candidato locale. Ogni confronto implica un possibile aggiornamento delle heap delle distanze vicine di questi punti.
2. Ogni candidato locale viene confrontato con gli oggetti candidati non locali. Per ogni confronto è possibile un aggiornamento della heap del solo oggetto locale.
3. Ogni candidato locale viene confrontato con tutti gli altri oggetti locali. Il calcolo della distanza tra i due avviene solo se almeno uno dei due oggetti può essere un outlier, ovvero se almeno uno dei due ha peso attuale ≥ lower bound attuale. Come nel primo caso, ogni confronto può portare all'aggiornamento delle heap dei due oggetti coinvolti.

Terminata questa fase di confronto, vengono scelti come nuovi candidati i primi m_i oggetti con peso più grande e rigorosamente ≥ lower bound. I candidati scelti verranno inseriti nella heap LC_i. Gli oggetti con peso ≤ lower bound vengono dichiarati non attivi.

A questo punto la procedura termina restituendo al nodo coordinatore il numero degli oggetti locali attualmente attivi, la struttura $LNNC_i$, contenente le heap delle prime *k* distanze per ogni candidato globale ricevuto, e la heap dei candidati locali da usare alla prossima iterazione, LC_i.

```
NodeCompi(minOUT, C, act, LNNCi, LCi, acti){
   load(acti, Ci);
   Di =  Di \ Ci;
   acti = acti - |Ci|;

   for each (p in Ci) {
      LNNCi[p] = NN[p];
   }

   for each (pj in Ci = {p1, … , p|ci|}) {
      for each (q in {pj, …, p|ci|}) {
            = dist(p, q);
         UpdateMin(LNNCi[pj], (q,   ));
         if (pj != q) UpdateMin(LNCCi[q], (pj,   ));
      }
   }

   for each (p in Ci) {
      for each (q in (C\Ci)) {
            = dist(p, q);
         UpdateMin(LNNCi[p], (q,   ));
      }
   }

   acti = 0;

   for each (p in Di) {
      for each (q in C) {
         if (max {Sum(NNi[p]), Sum(LNNCi[q])} >= minOUT) {
               = dist(p, q);
            UpdateMin(NNi[p], (q,   ));
            UpdateMin(LNNCi[q], (p,   ));
         }
      }
      if (Sum(NNi[p]) >= minOut) {
         acti = acti + 1;
      }
   }

   Ci = object(LCi);
   store(acti, Ci)
}
```

3.4 Correttezza dell'algoritmo

L'algoritmo determina i top-*n* outlier nel datataset distribuito *D*. Dato che gli oggetti restituiti come outlier sono stati confrontati con tutti gli altri oggetti del dataset distribuito, il peso che gli si viene attribuito è il peso reale. Per verificare la correttezza dell'algoritmo è sufficiente dimostrare che per ogni oggetto del dataset non facente parte della soluzione, il suo peso è inferiore o al limite uguale a quello dell'*n*-esimo outlier, ovvero all'ultimo lower bound.

Per dimostrarlo basta ricordarsi il funzionamento principale dell'algoritmo. L'algoritmo termina quando l'insieme dei candidati per la prossima iterazione è vuoto. Questo succede

quando ciascun insieme di candidati locali restituiti da ciascun nodo è vuoto, e quindi quando non ci sono più oggetti attivi. Un oggetto viene però etichettato come non attivo quando l'upper bound del suo peso è inferiore al peso dell'attuale n-esimo outlier. Di conseguenza questo oggetto non potrà mai essere un outlier.

3.5 Costo

Sia *a* il numero di attributi di ciascun oggetto del dataset, *t* il numero di iterazioni effettuate, *D* la dimensione del dataset globale e *l* il numero di nodi locali. O(*a*) denota il costo del calcolo della distanza tra due oggetti del dataset. La dimensione relativa del solving set rispetto alla dimensione del dataset distribuito è denotata come $\rho = \frac{|DSS|}{|D|} \leq 1$.

3.5.1 Costo temporale

La parte dominante dell'algoritmo sta nella procedura *NodeComp*, nella fase in cui i punti candidati vengono confrontati con i punti del dataset locale. Ciascun confronto costa O(*a*) per il calcolo della distanza, più il costo dell'eventuale aggiornamento delle heap dei punti coinvolti, O(log *k*). Queste due operazioni vengono eseguite O($m \cdot |D_i|$) volte, con *m* dimensione dell'insieme dei candidati e D_i la dimensione del dataset locale.

Facendo l'ipotesi che il dataset sia distribuito uniformemente lungo i nodi locali, il costo di ciascun nodo locale è: $O(tm \cdot \frac{|D|}{l}(a + \log k))$.

Dato che $tm = |DSS|$, il costo può essere riformulato come: $O(\frac{\rho}{l} \cdot |D^2|(a + \log k))$.

Un'osservazione importante è che, se non consideriamo il fattore ρ/*l*, abbiamo la complessità della strategia Naive Nested Loop, che calcola la distanza per tutte le coppie di oggetti presenti del dataset. L'algoritmo DistributedSolvingSet migliora il costo grazie all'introduzione del fattore ρ/*l* nel peggior caso possibile.

Riguardo al nodo coordinatore, le fasi dominanti sono quella in cui si determinano i *k* nearest neighbor per ogni candidato globale, e l'aggiornamento della heap dei top outlier. Facendo l'ipotesi che le $l \cdot k$ distanze vengano ordinate in modo da determinare le prime *k* distanze più piccole, il costo totale del coordinatore è:

$O(tm\cdot(kl\cdot\log(kl)+\log n))=O(\rho|D|\cdot(kl\cdot\log(kl)+\log n))$. Questo costo è ammortizzato dal fattore ρ.

Il costo temporale totale è ottenuto come somma dei due costi. Maggiore è il numero dei nodi locali e maggiore sarà il costo in carico ai nodi rispetto a quello in carico al supervisore.

In generale, con l'aumentare nel numero di nodi locali *l*, il costo in carico ai nodi locali sarà sempre più dominante sul costo sul nodo supervisore. Tuttavia, se il numero di nodi locali è così grande che il tempo di esecuzione assoluto dell'algoritmo è molto piccolo, allora i tempi di esecuzione del supervisore e dei nodi locali si equivalgono, e il tempo totale di esecuzione peggiora.

3.5.2 Costo di trasmissione

Se consideriamo il costo di trasmissione, inteso come quantità di dati che viene trasferita durante l'esecuzione dell'algoritmo, la comunicazione tra nodo supervisore e nodi locali è concentrata nelle procedure *NodeInit* e *NodeComp*. La prima è eseguita su ogni nodo locale una e una sola volta, e consiste nel trasferimento del numero intero m_i dal nodo supervisore a tutti i nodi locali, e nella successiva ricezione di m_i oggetti da ogni nodo locale. La seconda viene eseguita a ogni iterazione, quindi in tutto *t* volte, e consiste nel trasferimento dal nodo supervisore verso tutti i nodi locali di: un numero reale, *minOUT*, *m* oggetti candidati, contenuti in *C*, e un numero intero che rappresenta il numero di oggetti attivi, *act*. Ogni nodo locale invierà al nodo supervisore $m\cdot k$ distanze, contenute in $LNNC_i$, m_i oggetti candidati, contenuti in LC_i, e un intero act_i rappresentante il numero di oggetti locali rimasti attivi.

Ricordando che *a* rappresenta il numero di attributi di ciascun oggetto, il numero totale di dati trasferiti, TD, in termini di numeri interi o floating point è:

$$TD=\sum_{i=1}^{l}(1+m_i a)+t(1+ma+1+\sum_{i=1}^{l}(mk+m_i a+1))=l+ma+2\mathrm{t}+|DDS|(a+lk+a)+tl.$$

in quanto: $\sum_{i=1}^{l} m_i=m$ e $tm=|DSS|$.

Dato che si ha $m<<|DSS|$, e così, $am<<|DSS|a$, anche i termini *l*, *2t* e *2tl* sono trascurabili. Così, TD può essere approssimato a: $TD\approx|DSS|(lk+2\mathrm{a})$. Ora, ricordando che ρ=|DSS|/|D|, consideriamo il rapporto tra la quantità di dati trasferiti TD e la dimensione del dataset in termini di numeri floating point, $d\cdot a$: $TD\%=\frac{TD}{|D|a}\approx\frac{\rho lk}{a}$.

Da qui si nota che TD% è direttamente proporzionale a ρ, ovvero la dimensione relativa del solving set rispetto alla dimensione del dataset distribuito. Fissati i parametri *n* e *k* è stato osservato che ρ diminuisce più che linearmente rispetto all'aumentare della dimensione del dataset, |D|, perché la dimensione del solving set tende a stabilizzarsi.

Una osservazione importante è che anche se gli oggetti in esame hanno alta dimensionalità, intesa come numero di proprietà, la maggior parte delle informazioni trasferite sono comunque distanze, e non oggetti. Infatti TD% è inversamente proporzionale alla dimensionalità degli oggetti del dataset. Inoltre gli oggetti che vengono trasmessi sono solo quelli che appartengono al solving set, e rappresentano quindi una minima percentuale degli oggetti del dataset globale.

Tuttavia, la quantità di dati trasferiti aumenta linearmente con il numero dei nodi locali impiegati nella risoluzione del problema. Questo è un fattore da non sottovalutare, perché potrebbe portare a un peggioramento delle performance in uno scenario con un numero di nodi troppo alto e con un canale di comunicazione di velocità non adeguata alla quantità di informazioni da trasferire. Per ridurre questa dipendenza, l'algoritmo DistributedSolvingSet è stato rianalizzato ed è stata introdotta una sua variante, il LazyDistributedSolvingSet, che verrà esaminato nel prossimo capitolo e la cui idea è quella di trasferire inizialmente un numero limitato di distanze e, solo quando queste non sono sufficienti a calcolare il peso reale di un oggetto, trasferirne delle altre.

4 L'algoritmo LazyDistributedSolvingSet

Come introdotto alla fine del capitolo precedente, nell'algoritmo DistributedSolvingSet la quantità di informazioni inviate cresce linearmente col numero dei nodi impiegati nella risoluzione del problema. Il LazyDistributedSolvingSet limita la quantità di distanze inviate dai nodi locali adottando una strategia incrementale, composta da diverse iterazioni in cui viene inviato solo un piccolo insieme di distanze, a partire da quelle più piccole. Il nodo supervisore riceve così ogni volta un numero limitato di distanze e le utilizza per calcolare il peso dei candidati attuali; se le distanze in suo possesso non sono sufficienti a determinare il peso reale di alcuni candidati, viene eseguita una richiesta ai nodi locali e il nodo supervisione riceverà così nuove distanze per i soli candidati il cui peso reale è ancora incerto. Quando il nodo supervisore è in grado di calcolare il peso reale di ogni oggetto candidato, la procedura incrementale termina.

In questo modo il nodo coordinatore riceve da ogni nodo locale N_i un numero $k_i<k$ di distanze per ogni iterazione dell'algoritmo, per un totale di $l \cdot k_i$, O(k), valori. Questo consente di sostituire il termine $l \cdot k$ con k nel calcolo di TD e di limitare la dipendenza da l sulla quantità delle informazioni scambiate, che può essere approssimata a:

$$TD\% \approx \frac{\rho k}{a}.$$

Inoltre, la complessità temporale delle operazioni svolte dal nodo supervisore diventa:

$$O(\rho |D| \cdot (k \cdot \log(k) + \log n)).$$

4.1 Passi dell'algoritmo

Il comportamento è molto simile a quello dell'algoritmo DistributedSolvingset, soprattutto all'inizio. Anche qui, inizialmente il solving set e il solution set sono entrambi vuoti e il lower bound è fissato a 0, mentre il numero totale di nodi attivi è inizializzato alla dimensione del dataset globale.

Il coordinatore effettua la chiamata a una procedura di inizializzazione, *NodeInit()*, che verrà eseguita su tutti i nodi locali. Ogni nodo locale inizializzerà le proprie strutture e sceglierà casualmente m_i oggetti candidati dal suo dataset, che invierà al coordinatore.

Il coordinatore crea l'insieme dei candidati globali raggruppando gli oggetti ricevuti dai nodi locali e comincia la fase iterativa. Alla fine di ciascuna fase viene eletto un nuovo insieme di candidati per la prossima iterazione. Se questo insieme è vuoto l'algoritmo termina.

In ogni iterazione, per prima cosa il coordinatore aggiunge l'insieme corrente dei candidati al solving set, poi chiede ai nodi locali di eseguire la procedura di computazione, *NodeComp()*. In questo caso questa procedura è leggermente diversa da quella originale, perché i nodi locali ricevono in più un nuovo parametro, $k_0 < k$, che rappresenta il numero delle prime distanze vicine da restituire per ogni candidato. Così, ogni struttura $LNNC_i$ inviata dal nodo locale N_i al supervisore conterrà per ogni candidato un numero limitato di distanze, $k_0 = \lceil k/l \rceil + 1 < k$.

Il coordinatore utilizza le informazioni ricevute per aggiornare il numero globale di nodi attivi e per calcolare per ogni candidato il suo peso sulla base delle distanze ricevute. Questa seconda operazione è svolta tramite una procedura incrementale, in cui, per ogni candidato q:

- si raccolgono le distanze ricevute e si aggiorna la collezione $NNC[q]$ delle prime k distanze più prossime;
- se tutte le distanze in $NNC[q]$ risultano essere minori dell'ultimo elemento di ogni $LNNC_i[q]$, chiamato da ora in poi $lastLNNC_i[q]$ e che rappresenta la più grande distanza che il nodo locale N_i ha inviato per il candidato q, allora $NNC[q]$ contiene le k distanze più prossime per il candidato q, e quindi il peso di q calcolato sulla base di quelle distanze sarà il suo peso reale. Questo è vero perché ogni nodo locale inserisce in $LNNC_i$ le distanze ordinate in maniera crescente, di conseguenza le distanze non ancora inviate saranno sicuramente più grandi, o al limite uguali, a $lastLNNC_i[q]$. La procedura incrementale può quindi terminare;
- altrimenti, definiamo $dist_{min} = \min_i\{lastLNNC_i[q]\}$, ovvero la più piccola distanza tra le maggiori distanze di tutti gli heap $LNNC_i[q]$ ricevuti. $dist_{min}$ sarà presente in $NNC[q]$ in una certa posizione j, e le prime j distanze in $NNC[q]$ saranno quelle che separano q dai rispettivi primi j neareast neighbor, mentre le rimanenti $k - j$ rappresentano solo un upper bound delle vere distanze che separano q dai successivi $k - j$ neareast neighbor. Il numero di queste distanze "sconosciute" viene memorizzato nella collezione $u\_NNC[q]$, mentre nella collezione $cur\_last[q]$ viene memorizzato l'upper bound della distanza di q dal k-esimo nearest neighbor, $NNC[q][k]$, e nella collezione $nodes[q]$ si tiene traccia di quali sono i nodi che possono fornire almeno una distanza utile per tale candidato.

A questo punto, per ciascun candidato il cui peso è incerto, il nodo supervisore, tramite la procedura *NodeReq*(), richiede nuove distanze ai soli nodi locali precedentemente etichettati come "utili" per tale candidato. A ciascun nodo locale invia quindi le informazioni raccolte precedentemente e si aspetta di ricevere una nuova collezione di distanze. In particolare, ogni nodo locale invierà al più le successive $\lceil u\_NNC[q]/|nodes[q]| \rceil + 1$ distanze più piccole, e in ogni caso solo quelle con valore inferiore all'upper bound specificato da *cur_last*[*q*].

Ricevute le nuove distanze si dà il via a una nuova iterazione della procedura incrementale, in cui il nodo supervisore aggiornerà, per ciascun candidato *q*, la lista *NNC*[*q*] delle prime *k* distanze e determinerà se tali distanze sono sufficienti o meno a calcolare il peso reale del candidato ed eventualmente si eseguiranno nuove richieste ai nodi locali, e così via finché i pesi reali di tutti gli oggetti candidati non saranno stati calcolati.

Una volta terminata la procedura incrementale si conoscono i pesi reali di tutti i candidati attuali e si può procedere con l'aggiornamento della heap dei top-*n* outlier: anche qui, se un candidato ha un peso maggiore dell'attuale *k*-esimo outlier, viene aggiunto alla heap degli attuali top-n e viene calcolato il nuovo lower bound. L'ultimo passo dell'iterazione consiste nel raccogliere i candidati restituiti da ciascun nodo locale tramite la *NodeComp*(), creando così l'insieme dei candidati che verranno esaminati nella prossima iterazione. Se questo insieme è vuoto, l'algoritmo termina.

Pseudo codice

```
DistributedSolvingSet(l, [d1, … , dl], dist, n, k, m) {
   DSS = Ø;
   OUT = Ø;
   d =  i di;

   for each node Ni in N
      NodeInit(mi, Ci);

   C =  i Ci;
   act = d;
   minOUT = 0;

   while (C != Ø) {
      DSS = DSS    C;
      for each node Ni in N
         NodeComp(minOUT, C, act, ⌈k/l⌉+1 , LNNCi, LCi, acti);

      act =  i acti;

      repeat
```

```
         for each q in C {
            NNC[q] = get_k_NNC(NNC[q]     ( i LNNCi[q]));
            u_NNC[q] = 0;
            nodes[q] = Ø;
            if   j s.t. mini{last(LNNCi[q])} = NNC[q][j] {
               u_NNC[q] = k – j;
               cur_last[q] = NNCc[q][k];
               nodes[q] =   i Ni s.t. last(LNNCi[q]) in NNC[q];
            }
         }

         for each node Ni in   q in C nodes[q] {
            NodeReq(u_NNC, cur_last, nodes, LNNCi);
         }

      until   q in C nodes[q] = Ø;

      for each q in C {
         UpdateMax(OUT, <q, Sum(NNC[q])>);
      }

      minOUT = Min(OUT);
      C = Ø;
      for each p in   i LCi
         C = C   {p}
}
```

4.2 Procedure eseguite sui nodi locali

Da un punto di vista delle operazioni eseguite sui nodi locali, questo algoritmo introduce una piccola modifica alla procedura *NodeComp* e introduce la nuova procedura *NodeReq*.

NodeComp

La procedura *NodeComp* rimane sostanzialmente invariata eccetto che per la parte finale, in cui il nodo locale per ogni candidato non invia k distanze ma solo le prime k_0, mentre conserva le $k - k_0$ distanze rimanenti nella collezione *rLNNC*, in modo da averle già pronte nel caso il nodo supervisore gli chieda nuove distanze tramite la procedura *NodeReq*.

```
...

   LNNCi = sort(LNNCi);
   rLNNCi = get_l(LNNCi, k-k0);
   LNNCi = get_f(LNNCi, k0);
   store(acti, Ci, rLNNCi);
}
```

NodeReq

Per ogni candidato il cui nodo supervisore ha richiesto nuove distanze, il nodo locale che esegue la procedura *NodeReq* estrae dalla collezione delle distanze rimanenti, rLNNC, le prime $\lceil \frac{u\_NNC[q]}{|nodes[q]|} \rceil + 1$ distanze più piccole di valore inferiore all'upper bound specificato dal parametro *cur_last*. Queste distanze vengono inserite nella collezione *LNNC* che verrà restituita al nodo supervisore mentre, ancora una volta, le distanze rimanenti verranno conservate in *rLNNC* a fronte di eventuali richieste future.

Questa procedura ha un basso costo computazionale, dato che è molto semplice e non prevede il calcolo di nessuna distanza, e ha quindi un impatto trascurabile sui tempi di esecuzione finali.

```
NodeReqᵢ(u_NNC, cur_last, nodes, LNNCᵢ) {
   load(rLNNCᵢ);
   for each (q s.t. Nᵢ in nodes[q]){
      LNNCᵢ[q] = get_f(rLNNCᵢ[q], ⌈u_NNC[q] / |nodes[q]|⌉+1 , cur_last[q]);
      rLNNCᵢ[q] = get_l(rLNNCᵢ[q], |rLNNCᵢ[q]| - |LNNCᵢ[q]|);
   }
   store(rLNNCᵢ);
}
```

5 MPI, Message Passing Interface

MPI è una specifica di interfaccia per lo scambio di messaggi. L'obiettivo primario di MPI è di stabilire un efficiente, flessibile e portabile standard per la comunicazione a scambio di messaggi. Nonostante non sia uno standard IEEE o ISO è, di fatto, lo standard usato nello sviluppo di applicazioni basate sul message passing, destinate ad una esecuzione su piattaforme HPC, High Performance Computing.

Preso singolarmente non è quindi una libreria, ma un insieme di specifiche di come dovrebbe essere una libreria. Esistono diverse implementazioni che si differenziano nella versione e nelle features dello standard supportate.

5.1 Breve storia di MPI

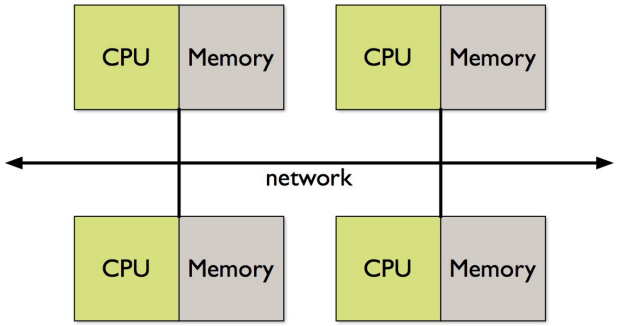
Prima degli anni '90, scrivere applicazioni parallele per diverse architetture era un lavoro più difficile di quanto lo sia oggi. Molte librerie facilitavano il processo, ma non c'era ancora una via standard per farlo.

A quei tempi, la maggior parte delle applicazioni parallele erano destinate ad ambienti di ricerca scientifica. Il modello più adottato dalle varie librerie era il message passing model, in cui la comunicazione tra i processi avviene tramite lo scambio di messaggi e senza l'uso di risorse condivise. Ad esempio, il processo master può assegnare un particolare lavoro agli slave semplicemente inviandogli un messaggio che gli descrive il lavoro da fare. Un secondo esempio molto semplice è quello di un'applicazione parallela che realizza un merge sort[2]: i dati vengono ordinati localmente ai processi e i risultati vengono passati ad altri processi che si occuperanno di fare il merge.

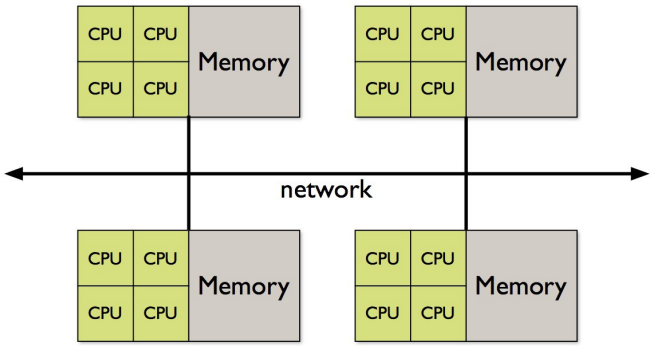
Dato che le librerie usate usavano sostanzialmente lo stesso modello, seppur con differenze minori l'una dall'altra, gli autori delle varie librerie si riunirono nel 1992 per definire una interfaccia standard per lo scambio di messaggi, e da qui nacque MPI, la Message Passing Interface. Questa interfaccia doveva permettere ai programmatori di scrivere applicazioni parallele portabili sulla maggior parte delle architetture parallele, usando le stesse features e i modelli a cui erano già abituati a usare.

2 Merge sort: algoritmo di ordinamento basato su confronti che sfrutta l'approccio del Divide et impera, “dividi e conquista”.

In origine, MPI è stato pensato per architetture a memoria distribuita, che cominciarono ad aumentare la loro popolarità una ventina di anni fa.



Col tempo, i sistemi a memoria distribuita hanno cominciato ad essere combinati tra loro, creando dei sistemi ibridi a memoria distribuita/condivisa. Gli implementatori hanno così adattato le loro librerie per gestire entrambi i tipi di architettura in maniera seamless.



Oggi, MPI viene eseguito su sistemi a memoria distribuita, a memoria condivisa, e ibridi. Il modello di programmazione resta comunque quello della memoria distribuita, nonostante la vera architettura su cui si esegue il calcolo possa essere diversa.

I punti di forza di MPI possono essere riassunti in:

- Standardizzazione: è supportato da tutte le piattaforme di High Performance Computing.
- Portabilità: le modifiche da applicare al codice sorgente sono nulle, o minime, nel caso si decida di portare l'applicazione su una piattaforma diversa, ma che supporta anch'essa lo stesso standard.
- Performance: i produttori possono creare implementazioni ottimizzate per un certo tipo di hardware e ottenere migliori performance.
- Funzionalità: oltre 440 routine definite in MPI-3, ma molti programmi paralleli possono essere scritti utilizzandone anche meno di 10.

5.2 Concetti chiave

5.2.1 Communicator e rank

Un communicator definisce un gruppo di processi che hanno la capacità di comunicare tra loro. La maggior parte delle routine richiedono di specificare il communicator tra gli argomenti. MPI_COMM_WORLD è il communicator predefinito e include tutti i processi.

L'identificazione di un processo è basata sui rank. A ogni processo viene assegnato un rank per ogni communicator di cui fa parte. Il rank è un numero intero che viene assegnato a partire da zero, e identifica ogni singolo processo nel contesto di uno specifico communicator. La prassi comune è quella di definire il processo avente rank globale 0 come il processo master. Tramite il rank lo sviluppatore può specificare qual è il processo mittente e quali sono invece i processi destinatari.

5.2.2 Comunicazione

La comunicazione è basata sulle operazioni di invio e ricezione tra i processi. Un processo può inviare un messaggio ad un altro specificando il rank del processo destinatario e il comunicatore al cui il rank fa riferimento. Il processo ricevente può quindi usare l'operazione di ricezione, specificando il rank del processo da cui si aspetta di ricevere il messaggio e, anche lui, il comunicatore. Una comunicazione di questo tipo è chiamata *point-to-point communication*.

In molti casi vi è la necessità di mettere in comunicazione un certo processo con tutti gli altri, ad esempio quando il processo master deve inviare delle informazioni a tutti i processi slave. In questo caso, una comunicazione point-to-point sarebbe decisamente scomoda. MPI gestisce una varietà di tipi di comunicazione collettiva in grado di coinvolgere tutti i processi.

Questo mix di operazioni, point-to-point e comunicazioni collettive, possono essere usate per creare programmi paralleli anche molto complessi, senza però complicare troppo la vita dello sviluppatore.

5.3 Inizializzazione dell'environment MPI

Di seguito un semplice Hello World in C, che illustra l'utilizzo delle principali primitive di gestione che ogni applicazione deve includere.

```
#include <mpi.h>;

int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(int *argc, char ***argv);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d"
         " out of %d processors\n",
         processor_name, world_rank, world_size);

  // Finalize the MPI environment.
  MPI_Finalize();
}
```

Il primo step è quello di includere il file header di MPI, con #include <mpi.h>.

Dopodiché, l'ambiente MPI deve essere inizializzato tramite MPI_Init. Questa routine deve essere chiamata prima di qualunque altra chiamata a una primitiva MPI, e deve essere chiamata una sola volta. Durante questa operazione vengono create e inizializzate tutte le proprietà dell'ambiente MPI. Ad esempio, viene assegnato un rank a ciascuno dei processi e viene creato un communicator globale che li racchiude tutti (MPI_COMM_WORLD). MPI_Init prende in ingresso due argomenti che non sono realmente necessari secondo lo standard, ma lo potrebbero essere a seconda dell'implementazione che si sta utilizzando.

MPI_Comm_size restituisce la dimensione del communicator. Nell'esempio, il comunicatore specificato racchiude tutti i processi, e quindi la dimensione restituita è uguale al numero dei processi allocati per l'esecuzione del programma.

MPI_Comm_rank restituisce il rank del processo chiamante, nel contesto del communicator specificato.

MPI_Finalize è usata per terminare e pulire l'ambiente MPI. Nessun'altra primitiva MPI può essere chiamata dopo questa istruzione.

5.4 Comunicazione point-to-point

Le operazioni point-to-point consistono nello scambio di messaggi tra due, e solo due, processi.

In un mondo perfetto, ogni operazione di invio sarebbe perfettamente sincronizzata con la rispettiva operazione di ricezione. Ovviamente non è questo il caso, e l'implementazione di MPI deve poter conservare i dati inviati quando i processi mittente e destinatario non sono sincronizzati. Tipicamente questo avviene tramite l'utilizzo di un buffer, trasparente allo sviluppatore e interamente gestito dalla libreria.

Le principali primitive di invio e ricezione sono le MPI_Send e MPI_Receive, ma lo standard specifica diverse loro varianti.

Send e receive

Send e receive sono i due concetti fondamentali di MPI. Quasi ogni funzione in MPI può essere implementata tramite queste due semplici primitive.

Le chiamate a send e receive operano nella seguente maniera. Prima, il processo A mette su un buffer tutto quello che vuole inviare al processo B chiamando la primitiva MPI_Send. Fatto questo, il mezzo di comunicazione è responsabile del routing del messaggio alla locazione di destinazione. La locazione di destinazione è specificata dal rank del processo B.

Il processo destinatario del messaggio è a conoscenza sia del fatto che deve ricevere un messaggio, sia da chi lo deve ricevere. Invocherà così la MPI_Receive con gli opportuni parametri.

Ci sono situazioni in cui il processo A può inviare diversi tipi di messaggi a B. Invece di richiedere a B di discriminare tutti questi messaggi, MPI permette di specificare un ID (tag) per ciascun messaggio.

MPI garantisce l'ordine nella ricezione dei messaggi: se un processo invia in sequenza due messaggi allo stesso destinatario, allora questo riceverà il primo messaggio con la prima receive, e il secondo messaggio con la seconda.

La maggior parte delle operazioni di comunicazione point-to-point possono essere utilizzate in maniera bloccante o non bloccante.

- Una send bloccante ritorna solo una volta che è sicuro modificare nuovamente il buffer. Sicuro significa che eventuali modifiche non possono avere alcun effetto alle

informazioni che il processo ricevente dovrà ricevere.

Una receive bloccante ritorna solamente dopo che l'informazione è stata ricevuta ed è pronta da utilizzare.

- La send e la receive non bloccanti funzionano in maniera simile: ritornano subito. Eseguono semplicemente una richiesta di esecuzione di una operazione, che verrà effettivamente eseguita non appena sarà possibile farlo. Non attendono quindi che l'operazione si completi.

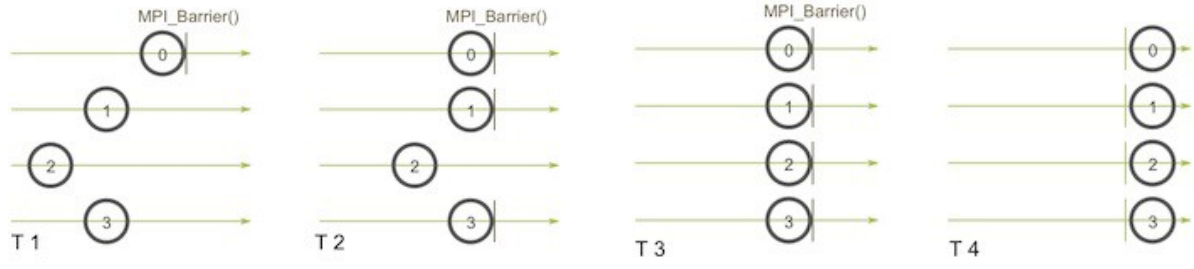
  Dato che non è sicuro modificare il buffer senza sapere se la richiesta è stata eseguita (ad esempio, il dato potrebbe ancora non essere stato ricevuto) la primitiva wait viene utilizzata allo scopo di bloccare il processo chiamante finché la send/receive non bloccante specificata non è stata completata.

  In molti casi, l'uso consapevole di una comunicazione non bloccante può avere un buon impatto sulle performance. Ad esempio, il mittente può eseguire delle operazioni di altro tipo mentre è in attesa che si concluda la trasmissione del messaggio.

5.5 Comunicazioni collettive

Le routine di comunicazione collettiva coinvolgono tutti i processi del communicator specificato. Una cosa importante da ricordare è che una comunicazione collettiva implica un punto di sincronizzazione per tutti i processi. Questo significa che tutti i processi devono aver raggiunto un certo punto prima che possano, tutti assieme, proseguire. Per fare questo, MPI definisce il concetto di barriera e la sua funzione omonima. Ogni primitiva di comunicazione collettiva utilizza implicitamente questa funzione.

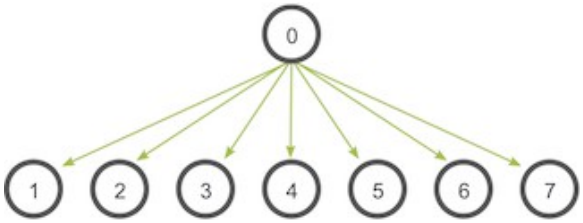
Lo scopo della funzione MPI_Barrier è abbastanza semplice: ci troviamo in uno scenario in cui ogni processo, arrivato ad un certo punto della sua esecuzione, invocherà tale funzione; nessuno di questi processi potrà proseguire finché tutti i processi del communicator non avranno raggiunto lo stesso punto, ovvero finché non avranno invocato tutti la funzione MPI_Barrier.



MPI definisce diverse tipologie di comunicazione collettiva. Le principali sono la broadcast, la scatter, la gather e la allgather.

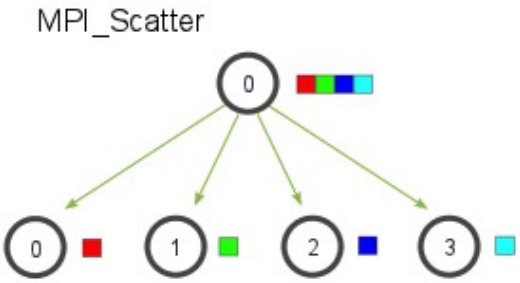
Broadcast

Il processo mittente invia una informazione che dovrà essere ricevuta da tutti gli altri processi del communicator specificato nel momento della chiamata a MPI_Bcast.



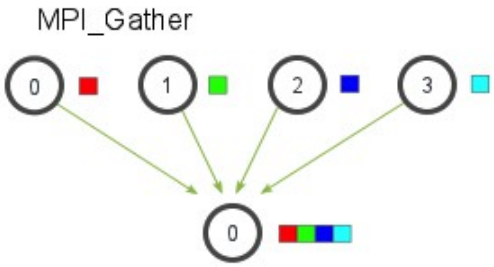
Scatter

Mentre con la broadcast tutti i processi ricevono la stessa informazione, tramite la scatter ciascun processo riceve un elemento diverso di un array. In particolare, il mittente specifica l'invio di un array e i suoi elementi verranno distribuiti tra i vari processi a seconda del loro rank: il processo di rank i riceverà l'elemento situato nella posizione i-esima.



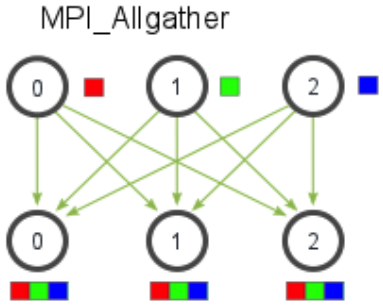
Gather

Una gather è sostanzialmente l'inverso di una scatter. I messaggi inviati dai vari processi hanno un tutti lo stesso destinatario.



Allgather

Ogni processo invia il suo messaggio a tutti gli altri processi, e riceve i messaggi inviati da questi.



5.6 Tipi di dato

Ogni volta che si usa una primitiva per lo scambio di un messaggio, è necessario specificare anche il tipo del dato coinvolto nel trasferimento. MPI definisce diversi tipi primitivi come char (MPI_Char), int (MPI_Int), long, long long, float, e double.

Nel caso si volessero inviare informazioni di un tipo non primitivo, MPI offre diverse funzioni per la creazione di un tipo di dato derivato come composizione di dati primitivi, tra cui:

- MPI_Type_contiguous, per la creazione di un tipo di dato composto da n copie di un dato primitivo;
- MPI_Type_struct, per creare un nuovo tipo di dato a partire da un "struct" C-like composta da soli tipi primitivi.

```
Esempio di utilizzo del costruttore MPI_Type_struct.
Viene creato un tipo car_s composto da due proprietà di tipo int.
```

```
#include <mpi.h>
…

typedef struct car_s {
        int shifts;
        int topSpeed;
} car;

int main(int argc, char **argv) {

…

   /* create a type for struct car */
   const int nitems=2;
   int          blocklengths[2] = {1,1};
   MPI_Datatype types[2] = {MPI_INT, MPI_INT};
```

```
   MPI_Datatype mpi_car_type;
   MPI_Aint     offsets[2];

   offsets[0] = offsetof(car, shifts);
   offsets[1] = offsetof(car, topSpeed);

      MPI_Type_create_struct(nitems, blocklengths, offsets, types,
&mpi_car_type);
   MPI_Type_commit(&mpi_car_type);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0) {
      car send;
      send.shifts = 4;
      send.topSpeed = 100;

      const int dest = 1;
      MPI_Send(&send, 1, mpi_car_type, dest, 1, MPI_COMM_WORLD);
   }

   if (rank == 1) {
      MPI_Status status;
      const int src = 0;

      car recv;
          MPI_Recv(&recv, 1, mpi_car_type, src, 1 MPI_COMM_WORLD,
&status);
   }

…
```

6 Implementazione dell'algoritmo

L'algoritmo è stato implementato in C++ a partire da una sua versione Java precedentemente sviluppata. La scelta di una sua traduzione in C++ è stata vincolata dalla implementazione di MPI installata sulla macchina su cui si saranno dovuti fare i test. L'implementazione trovata su tale macchina è MPICH2, che supporta i linguaggi C, C++ e Fortran. Tra i tre si è quindi scelto C++ per via della sua natura object oriented.

6.1 Interazione master e slave

Le classi ODPMasterAlgo e ODPSlaveAlgo contengono i passi degli algoritmi lato master e lato slave. La comunicazione tra master e slave è affidata a due classi "proxy": MasterSocketProxy e SlaveSocketProxy.

L'esecuzione lato master è incentrata sul metodo exec() della classe ODPMasterAlgo. Durante l'esecuzione di questo metodo, il master chiama più volte i metodi del suo oggetto proxy, i quali inviano (MPI.broadcast) un particolare comando agli slave e restano poi in attesa (MPI.gather) dei risultati ottenuti da ciascuno di essi.

L'esecuzione lato slave è invece incentrata sul metodo broadcast() della classe SlaveSocketProxy. In questo metodo il processo slave attende ciclicamente (MPI.broadcast) la ricezione di un comando; a seconda del comando ricevuto viene invocato uno specifico metodo dell'oggetto ODPSlaveAlgo, il quale calcola e restituisce un risultato che il proxy invierà (MPI.gather) al processo master.

I comandi inviati dal master agli slave consistono in:

- Invio dei parametri iniziali che rappresentano il problema, tra cui: n, k, m, r e l'identificativo funzione distanza da utilizzare. Il compito degli slave sarà quello di estrarre le informazioni ricevute e usarle per inizializzare l'oggetto ODPSlaveAlgo. Infine invieranno un ack di conferma al master.
- Richiesta della cardinalità del dataset distribuito. Ciascuno slave dovrà inviare al master la cardinalità del suo dataset.
- Richiesta dei candidati iniziali. Ogni slave invierà un vettore di candidati scelti in

maniera casuale. Il proxy lato master li riunirà tutti in un unico vettore e lo restituirà al chiamante.

- Richiesta di computazione. Gli slave ricevono un oggetto contenente i candidati globali, il lowerbound, il numero dei punti attivi e il numero di vicini da considerare. Il loro compito sarà confrontare questi punti sia tra di loro, sia con tutti gli altri punti appartenenti al dataset locale, in modo da ricavare il nuovo insieme dei candidati locali.

  Questo insieme verrà inviato al master assieme a un vettore che conterrà, per ciascun candidato globale ricevuto, la heap dei rispettivi nearest neighbor, e assieme ad altre informazioni come il numero delle distanze calcolate e il numero dei punti locali rimasti attivi.

- Richiesta di nuove distanze per i candidati globali. Inviata quando il master ha bisogno di ulteriori distanze per determinare il vero peso associato ai candidati. Ciascuno slave invierà così un oggetto contenente i vettori delle nuove distanze per i candidati specificati.

- Richiesta di terminazione. Inviato una volta individuati gli outlier, il compito degli slave è raccogliere le loro statistiche di esecuzione (tempo di esecuzione, distanze calcolate, distanze inviate, utilizzo massimo di memoria) e inviarle al master, che le utilizzerà per generare i report finali.

6.2 Punti e candidati

Punti e candidati, realizzati dalle classi Point e Candidate, sono molto simili ed estendono entrambi la classe Element, che definisce tutto ciò che hanno in comune.

Entrambi hanno un peso, un vettore di coordinate, e una stringa che li identifica in modo univoco e che specifica anche la loro appartenenza a uno specifico slave. In più i punti hanno la heap delle distanze dei loro k vicini.

Questa distinzione è voluta per limitare l'overhead di trasmissione, dato che la heap dei vicini è un'informazione utilizzata solo dagli slave. In questo modo master e slave non si scambiano mai oggetti Point ma sempre e solo oggetti Candidate, più leggeri. In particolare, quando è lo slave a inviare, converte i suoi oggetti Point nella rispettiva versione Candidate. Al contrario, quando lo slave riceve i candidati globali, deve individuare quali di questi appartengono al

dataset locale e lavorare sulle loro versioni Point, in modo da mantenere aggiornate le rispettive heap.

6.3 Heap

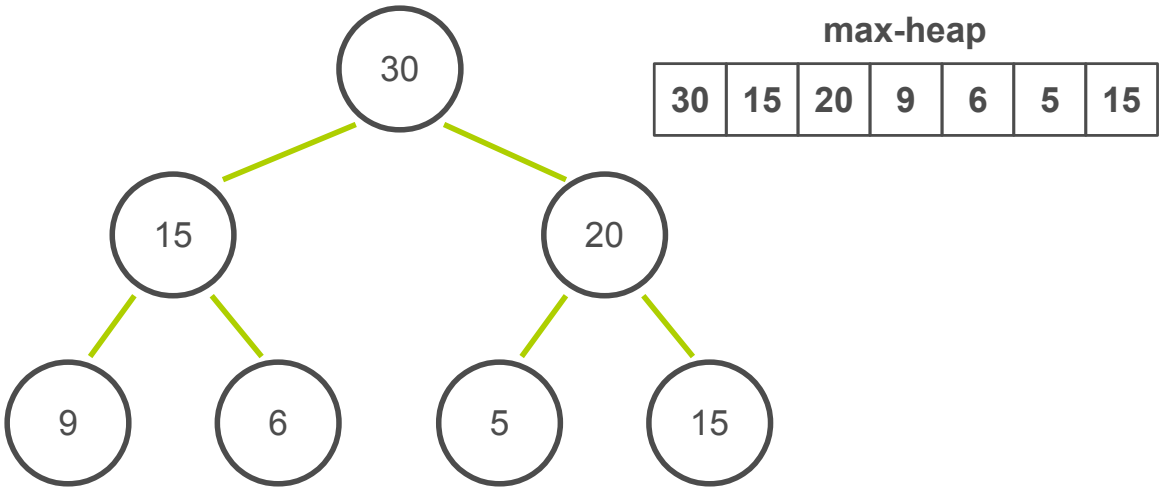
La heap è una particolare struttura dati ad albero binario che soddisfa la seguente proprietà:

- nel caso di una min-heap, fissata una certa funzione di comparazione, un qualunque elemento padre ha sempre un valore minore o al limite uguale al valore dei suoi figli;
- nel caso di una max-heap, l'elemento padre ha sempre un valore maggiore o al limite uguale al valore dei suoi figli.

Nonostante venga interpretata come un albero, è rappresentabile come un vettore di elementi in cui:

- il primo elemento è la radice dell'albero;
- il secondo e il terzo elemento sono i figli della radice;
- il quarto e il quinto elemento sono i figli del secondo elemento, e così via.

Quindi, il primo elemento è sempre l'elemento più grande, nel caso della max-heap, o è sempre il più piccolo, nel caso della min-heap.



Nel namespace core:elements:heaps, l'header “MyHeap.h” definisce una generica heap basata sui template e la collezione std::vector e i comparatori necessari per il mantenimento delle proprietà di heap. Nello stesso namespace troviamo poi le classi HeapP e HeapC, che realizzano delle min-heap rispettivamente di punti e di candidati, e la classe NN che realizza

la max-heap delle distanze di un punto dai suoi primi k vicini.

La scelta di organizzare gli elementi come delle heap è dovuta al fatto che si è sempre interessati solo all'elemento radice e non è quindi necessario avere un ordinamento assoluto degli elementi vettore, cosa molto più costosa. Per fare un esempio, ogni punto contiene un oggetto NN, che rappresenta la max-heap delle distanze del punto dai suoi primi k vicini. Quando tale punto viene confrontato con un altro punto, la sua heap viene aggiornata solo se tale distanza è inferiore all'elemento radice (il primo elemento del vettore), e quindi solo se è inferiore all'elemento più grande presente. La rimozione dell'elemento radice e l'inserimento del nuovo elemento, e in generale tutte le operazioni che effettuano una modifica sulla struttura, sono state implementate in modo da garantire sempre la proprietà di heap.

6.4 Organizzazione dei namespace

Il software è organizzato in diversi namespace:

- core: contiene le classi che definiscono i passi dell'algoritmo lato master e lato slave;
    - configurations: classi per la configurazione del master e dello slave;
    - elements: definisce la classe base Element e le sue due estensioni Point e Candidate;
        - heaps: implementazione delle minHeap di punti e candidati, definizione della nearest neighbor di un punto come una maxHeap delle sue prime k distanze;
    - instances: contiene le classi per la rappresentazione delle istanze del problema lato master e lato slave;
    - master: classi per la gestione dei report, delle statistiche e del solvingset;
    - net:elements: definisce le classi di tutti gli oggetti che verranno inviati al master da parte degli slave;
- commands: classi che definiscono gli oggetti inviati dal master agli slave e che verranno interpretati come comandi;
- extra: contiene la classe GetRSS, usata per ottenere la quantità di memoria usata dai processi;
    - function: classi per la definizione delle funzioni distanza. Per il momento è definita

solo la square distance;

    - logging: classi per la gestione del logging degli eventi;
    - timing: classi per tenere traccia del tempo di esecuzione;
- file
    - input: contiene le classi per la lettura dei dataset in formato .csv e la creazione degli oggetti Point a partire dalle entry lette;
    - output: classi per il salvataggio dei report come file di testo;
- masterproxy: definisce l'interfaccia che verrà implementata dalla classe MasterSocketProxy;
- socket: qui troviamo le classi proxy tramite le quali metteremo in comunicazione master e slave.

6.5 Le librerie Boost

Boost è un set di oltre 80 librerie open source per C++ che estendono le funzionalità di questo linguaggio. Tra le librerie presenti troviamo quelle per la gestione del file system, le espressioni regolari, il supporto al multithreading, l'elaborazione di immagini, serializzazione e MPI. Con questo, semplificano spesso la vita al programmatore fornendogli vie più semplici per eseguire determinati task.

Per l'implementazione dell'algoritmo è stata utilizzata la libreria Boost.MPI, che definisce così una sorta di "'interfaccia a un'interfaccia" che si sposa meglio con lo stile moderno di programmazione in C++.

Le ragioni che mi hanno condotto alla scelta di utilizzare questa libreria sono:

- Semplicità: quando si invoca una primitiva di comunicazione, bisogna specificare solamente gli argomenti più importanti: la variabile che contiene il messaggio da trasferire (o che conterrà il messaggio ricevuto), il comunicatore, il rank del mittente e il tag del messaggio. Non è necessario specificare né il tipo né la lunghezza del messaggio.
- Supporto a tipi di dato non primitivi: la libreria fornisce completo supporto ai data types della C++ Standard Library, ad esempio std::string e std::vector, ma anche a

quelli definiti dall'utente. In questo modo è possibile inviare qualunque messaggio in maniera del tutto trasparente, senza dover prima creare un tipo di dato derivato come avviene tipicamente.

Il supporto ai tipi è stato particolarmente utile perché i processi master e slave si scambiano messaggi di diversi tipi: da tipi primitivi, a stringhe, collezioni di tipi primitivi, ma soprattutto oggetti che contengono collezioni di oggetti.

Senza il supporto di questa libreria si sarebbe dovuto definire un data type derivato per ciascun oggetto coinvolto nel trasferimento, e quindi anche per ciascun tipo non primitivo contenuto in esso, e anche per ciascuna collezione presente. Così facendo, si sarebbero inviati dei messaggi che al loro interno racchiudevano sì le stesse informazioni degli oggetti, ma non i loro metodi. Lato destinatario sarebbe stato quindi necessario ricostruire gli oggetti a partire dai dati ricevuti, oppure utilizzare delle classi con metodi statici il cui comportamento era lo stesso dei metodi di cui si aveva bisogno.

L'alternativa alla creazione di tanti data type è quella di serializzare a monte l'oggetto da inviare e deserializzarlo a valle, ma questo è proprio quello che fa Boost ogni volta che viene invocata una primitiva di comunicazione.

6.5.1 Build delle librerie

La maggior parte delle librerie sono specificate interamente come file header basati sui template e sono quindi “pronte all'uso”. Altre librerie invece necessitano una compilazione.

Boost.MPI si appoggia alla libreria BOOST.Serialization, ed entrambe richiedono di essere compilate. Questa procedura è comunque molto veloce e si svolge in pochissimi passi. Inoltre non è necessario avere i privilegi di root.

Una volta scaricato ed estratto l'archivio della libreria:

- dal terminale, posizionarsi all'interno della cartella appena estratta;
- eseguire ./ bootstrap ;
- editare il file <cartella_estratta>/tools/build/v2/user-config.jam scrivendo nella prima riga “using mpi ;”, stando attenti agli spazi. A questo punto basta scegliere una cartella che conserverà i binari, riposizionarsi nella cartella principale ed eseguire: ./b2 install --with-mpi --with-serialization –libdir=<percorso_cartella>;

- per la compilazione dell'applicazione, aggiungere l'opzione: -I"<percorso_cartella>" ;
- per il link, invece, aggiungere: -L<percorso_cartella>/stage/lib -lboost_mpi -lboost_serialization

6.5.2 Utilizzo

Di seguito un breve confronto tra l'invocazione di funzioni della libreria Boost e l'invocazione della rispettiva primitiva MPI, limitatamente a quelle usate nell'implementazione dell'algoritmo.

Inizializzazione dell'environment

L'environment e il communicator hanno due costruttori di default che non richiedono nessun argomento. I metodi size() e rank() restituiscono rispettivamente la dimensione del communicator e il rank del processo nel contesto del communicator su cui viene invocato il metodo.

Inizializzazione dell'environment e creazione del communicator	
Boost.MPI	MPI

`#include <boost/mpi/environment.hpp>`
`#include <boost/mpi/communicator.hpp>`
…
 	
`int main()`
`{`
`  mpi::environment env;`
`  mpi::communicator world;`
`  int size = world.size();`
`  int rank = world.rank();`
 	
…
	`#include <mpi.h>`
	…
	
	`int main(int argc, char* argv[])`
	`{`
	`MPI::Init(argc, argv);`
	`int size=MPI::COMM_WORLD.Get_size();`
	`int rank=MPI::COMM_WORLD.Get_rank();`
	
	…

Send e receive

Le send e le receive sono metodi del communicator e richiedono la specifica di tre parametri:

- il rank del destinatario nel caso della send, del mittente nel caso della receive;
- il tag che identifica il messaggio;
- il valore o la variabile/oggetto contenente il valore da trasmettere, nel caso della send, e la variabile che conterrà il valore ricevuto del caso della receive. Non è necessario

specificare né il tipo né la dimensione del dato da inviare. L'unico requisito è che il tipo della variabile di destinazione sia dello stesso tipo del messaggio inviato.

Send e receive	
Boost.MPI	MPI

`int numberToSend = 10;`
`world.send(1, 0, numberToSend);`
 	
`…`
 	
`int numberToReceive;`
`world.recv(1, 0,`
`numberToReceive);`
 	
`…`
	`int number = 10;`
	`MPI::COMM_WORLD.Send(&numberToSend,`
	`1, MPI::INT, 0, 1 );`
	`…`
	
	`numberToReceive;`
	`MPI::COMM_WORLD.Recv(&numberToRecei`
	`ve,`
	`1, MPI::INT, j, 1,`
	`status );`
	`…`

Broadcast e gather

Il metodo broadcast richiede tre argomenti:

- il comunicatore su cui deve essere trasmesso il messaggio;
- il rank del processo mittente;
- il valore o la variabile contenente il valore da trasmettere, nel caso della send, e la variabile che conterrà il valore ricevuto del caso della ricevere.

Il metodo gather richiede quattro argomenti:

- il comunicatore su cui deve essere trasmesso il messaggio;
- il valore o la variabile contenente il valore da trasmettere;
- il vettore che verrà popolato con i valori raccolti da tutti i processi. Per i processi che non raccolgono, questo parametro può essere omesso;
- il rank del processo che effettuerà la raccolta dei messaggi.

Broadcast e gather	
Boost.MPI	MPI

`int number = 10;`
`broadcast(world, number, 0);`
 	
`…`
 	
`std::vector<int> all_numbers;`
`gather(world, number,`
`all_numbers, 0);`
	`int number = 10;`
	`MPI_Bcast(&number, 1, MPI::INT,`
	`MPI::COMM_WORLD, 0);`
	
	`…`
	
	`int gsize,sendarray[100];`
	`int root, *rbuf;`

```
MPI::Comm_size( comm, &gsize);

rbuf
=(int*)new(gsize*100*sizeof(int));

MPI::Gather( sendarray, 100,
MPI::INT,
     rbuf, 100, MPI::INT, root,
comm);
```

Invio di un dato non primitivo

Come già anticipato, l'invio di qualunque dato tramite queste librerie è del tutto trasparente.

Nel caso particolare del trasferimento di un oggetto, l'unica operazione da fare è definire un metodo di serializzazione all'interno della sua classe. Una cosa importante da ricordare è che, se la classe contiene oggetti di altri tipi, è necessario che anche nelle relative classi sia specificato un metodo di serializzazione.

Come esempio verrà mostrata una parte della definizione della classe Element:

```
#include <boost/serialization/access.hpp>
…

class Element {
…

protected:
      // vettore di coordinate reali
      std::vector<float> coordinates;

      /// id univoco del punto
      std::string id;

private:
      friend class boost::serialization::access;
      template<class Archive>
      void serialize(Archive &ar, const unsigned int version) {
            ar & coordinates;
            ar & id;
}

…
```

7 Risultati sperimentali

Per valutare le performance dell'algoritmo sono stati eseguiti una serie di test su diverse tipologie di dataset.

La piattaforma di testing utilizzata è *FERMI*, un IBM Blue Gene/Q presente presso il centro di calcolo del *CINECA*. *FERMI* è uno dei supercomputer più potenti al mondo. È composto da 10.240 sockets PowerA2, 1.6GHz di frequenza, ciascuno con 16 core, per un totale di 163.840 computing core e una peak performance di sistema di 2,1 PFlops. Ogni processore è dotato di 16Gbyte di RAM (1 GByte per core). Secondo la Top500[3], nel giugno del 2012 si trovava alla settima posizione; un anno dopo, giugno 2013, alla dodicesima.

I dataset considerati nella fasi di test sono:

- *10G2d*: un dataset di 10 milioni di oggetti bi-dimensionali, generati da una distribuzione normale.
- *G2d*: un dataset di 1 milione di oggetti bi-dimensionali, generati da una distribuzione normale.
- *G3d*: un dataset artificiale contenente 500,000 vettori tri-dimensionali, ottenuto dall'unione di oggetti di tre differenti distribuzioni normali 3-d, ognuna avente un differente vettore delle medie e la matrice unitaria come matrice di covarianza.
- *Poker*: un dataset reale di 1 milione di istanze di 10 attributi, ottenuto rimuovendo la class label dal dataset Poker-Hands disponibile presso l'UCI Machine Learning Repository.
- *2Mass*: un dataset reale di 1623376 istanze di 3 attributi, contenente dati ottenuti dal database 2MASS Survey Atlas Image Info del 2MASS Survey Scan Working Databases catalog, disponibile presso il NASA/IPAC Infrared Science Archive (IRSA).
- *Covtype*: un dataset reale di 581012 istanze di 10 attributi, ottenuto selezionando gli attributi quantitativi del dataset Covertype, disponibile presso l'UCI Machine Learning Repository.

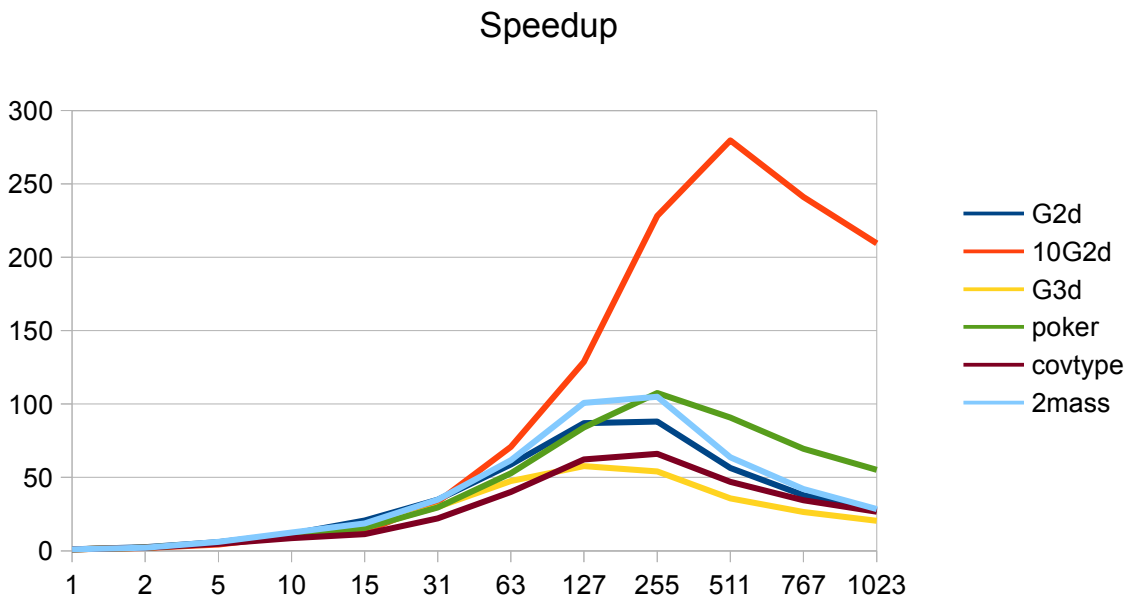
3 http://www.top500.org/

Se non specificato diversamente, la configurazione usata nella risoluzione dei problemi sarà sempre: $n = 10$, $k = 50$, $m = 100$.

Speedup e tempi di comunicazione

Per ciascun dataset è stato analizzato lo speedup dell'algoritmo al variare dei nodi locali impiegati. Lo speedup S_l è definito come il rapporto T_1/T_l, ovvero il rapporto tra il tempo di esecuzione dell'algoritmo eseguito con un solo nodo locale e il tempo di esecuzione ottenuto eseguendo lo stesso algoritmo con l nodi.

Dal grafico sottostante possiamo notare che, in generale, lo speedup aumenta notevolmente fino ad arrivare a una certa configurazione in cui si raggiunge un picco massimo. Se si insiste aumentando il numero dei nodi locali inizia un peggioramento.

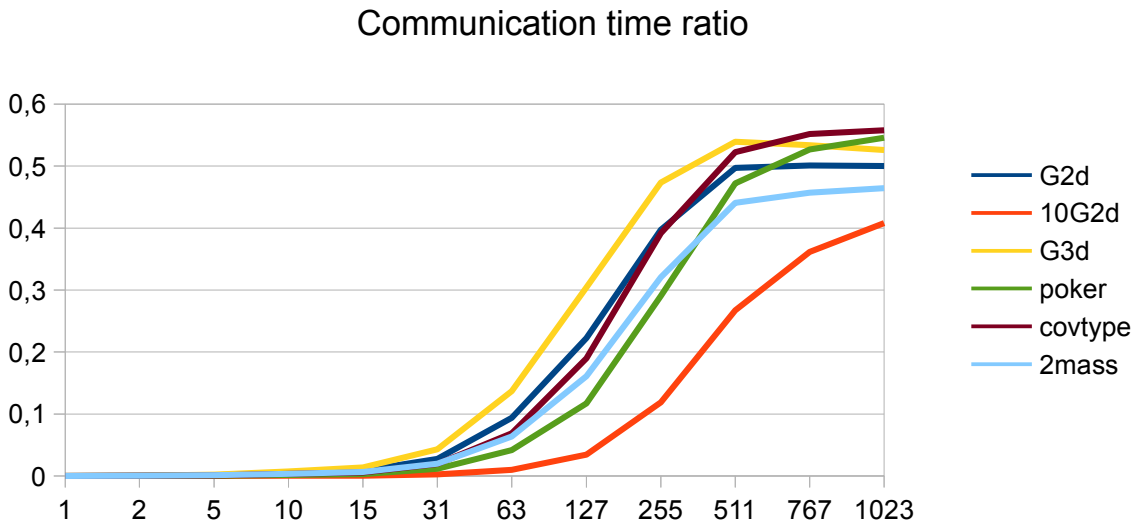


La principale causa di questo deterioramento dello speedup sembra essere il tempo impiegato nella comunicazione tra i nodi locali e il nodo supervisore. Infatti, all'aumentare del numero di nodi locali aumenta anche la quantità di informazioni che il nodo supervisore deve ricevere e, ovviamente, aumentano i destinatari delle informazioni che il nodo supervisore invia.

Il grafico seguente mostra il rapporto tra la stima del tempo di comunicazione e il tempo di esecuzione totale. Bisogna precisare però che la stima rappresenta in realtà un upper bound, perché in tale tempo vi è compreso anche il tempo necessario al trasferimento di informazioni necessarie alla post elaborazione, e quindi non strettamente correlate l'algoritmo in sé. Tuttavia possiamo notare, per qualunque dataset, quanto il tempo di comunicazione sia poco rilevante finché abbiamo un numero di nodi relativamente basso. In generale, una volta

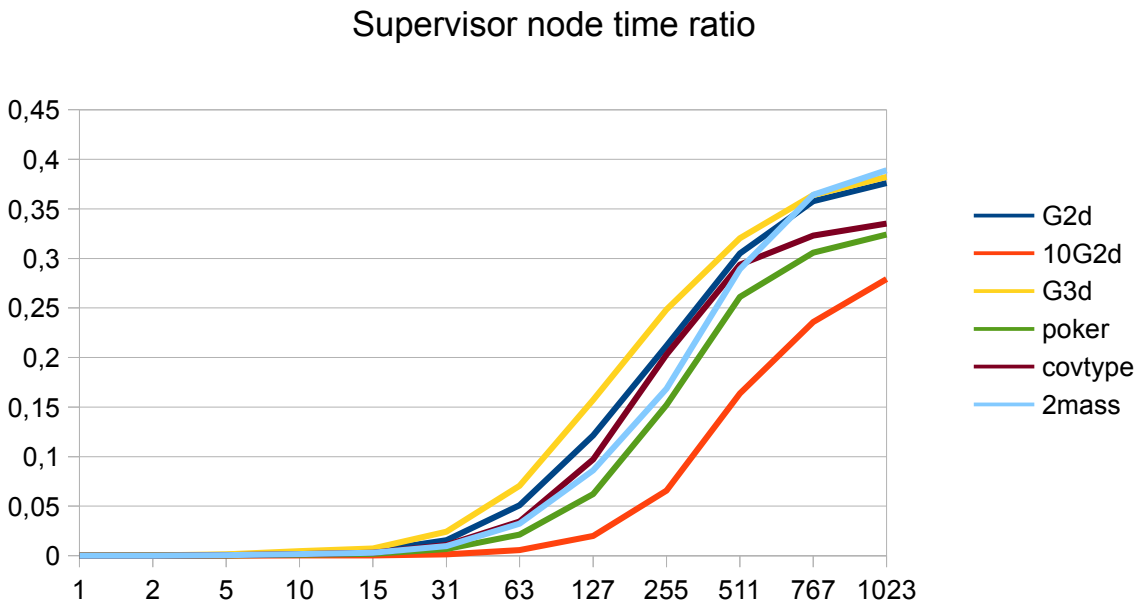
superato il “limite” dei 63 nodi locali, il tempo di comunicazione incide sempre di più, arrivando anche a toccare e superare il 50% del tempo di esecuzione totale. Questo limite dipende dalla natura del dataset e possiamo notare che un dataset molto grande come *10G2d*, composto da 10 milioni di oggetti, il tempo di comunicazione inizia a incidere più avanti rispetto agli altri dataset.

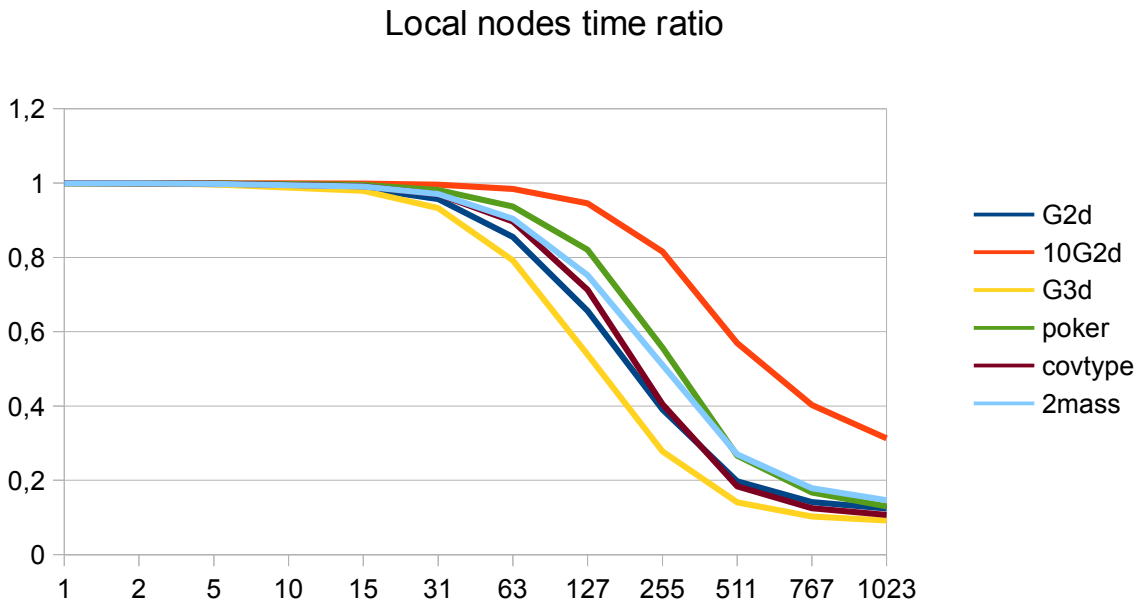
Quindi, utilizzando troppi nodi si arriva a una situazione in cui è più il tempo che si dedica alla comunicazione tra i nodi che il tempo di esecuzione effettiva sui nodi stessi.



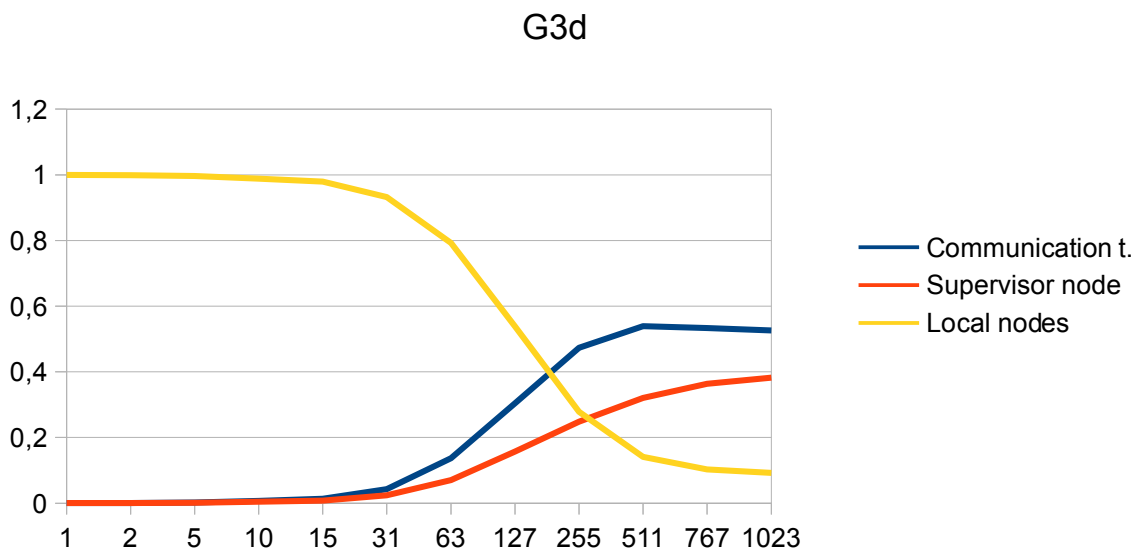
Come se non bastasse, con l'aumentare del numero di nodi anche la fase di sincronizzazione eseguita sul nodo supervisore inizia ad avere un suo peso. Il carico sui nodi locali diminuisce sempre di più, giustamente, ma raggiunto un certo numero di nodi il nodo coordinatore avrà un carico di lavoro che inciderà negativamente sulle performance.

Mostriamo quindi l'andamento del rapporto tra il tempo di esecuzione passato lato supervisore e il tempo di esecuzione totale, e il rapporto tra il tempo di esecuzione passato sui nodi locali e il tempo di esecuzione totale.


Supervisor node time ratio
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
1
2
5
10
15
31
63
127
255
511
767
1023
G2d
10G2d
G3d
poker
covtype
2mass


Local nodes time ratio
1,2
1
0,8
0,6
0,4
0,2
0
1
2
5
10
15
31
63
127
255
511
767
1023
G2d
10G2d
G3d
poker
covtype
2mass

Il grafico seguente mostra la distribuzione dei tempi di esecuzione al variare dei nodi per il dataset *G3d*.

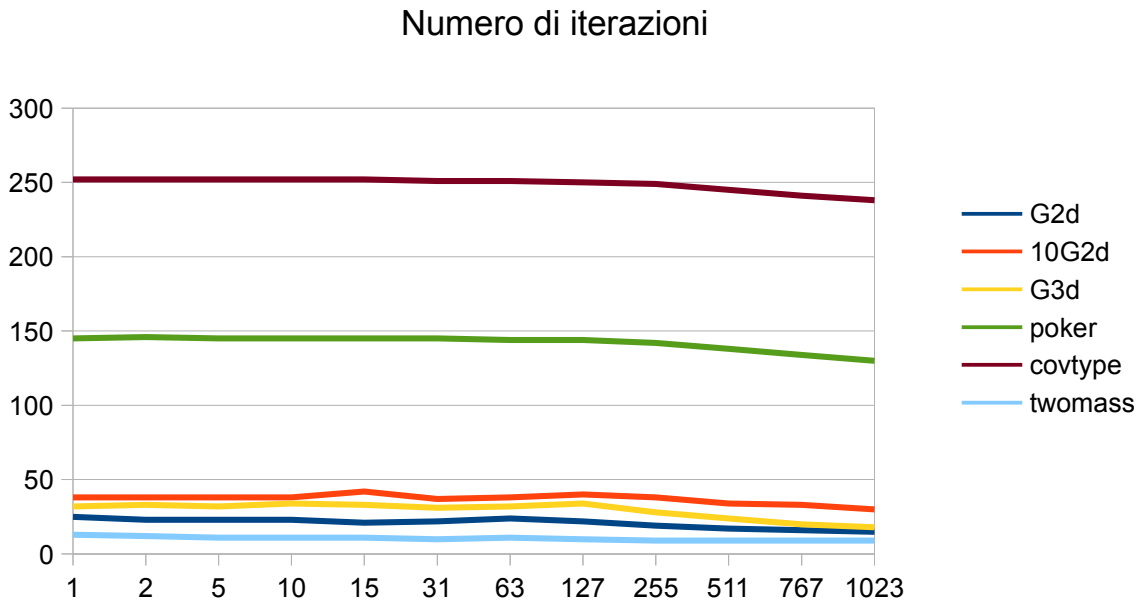


Cardinalità del solving set e numero di iterazioni

Confrontando la dimensione del solving set ottenuto dall'esecuzione con un solo nodo locale, (che ricordiamo, corrisponde all'esecuzione dell'algoritmo SolvingSet) con la dimensione dei solvingset ottenuti all'aumentare dei nodi locali impiegati, si può notare come in generale queste dimensioni differiscano di poco. Possiamo quindi dire che la qualità dei solving set ottenuti si equivalgono. La tabella mostra, per ogni dataset oggetto di studio, la dimensione assoluta e relativa del solving set ottenuto con $l = 1$, $l = 127$ e $l = 1023$.

Dataset	$l = 1$		$l = 127$		$l = 1023$	
	Abs. size	Rel. size	Abs. size	Rel. size	Abs. size	Rel. size
G2d	2391	0.002	2127	0.002	2323	0.002
10G2d	3701	0.0003	3927	0.0003	3824	0.0003
G3d	3117	0.006	3240	0.006	2623	0.005
Poker	14328	0.014	14331	0.014	13884	0.013
Covtype	25087	0.043	24908	0.042	24661	0.042
2Mass	1222	0.001	927	0.0009	1638	0.001

Riguardo al numero di iterazioni necessarie per il completamento dell'algoritmo, possiamo notare che è all'incirca costante e che comincia a diminuire solo quando il numero dei nodi locali impiegati è alto.

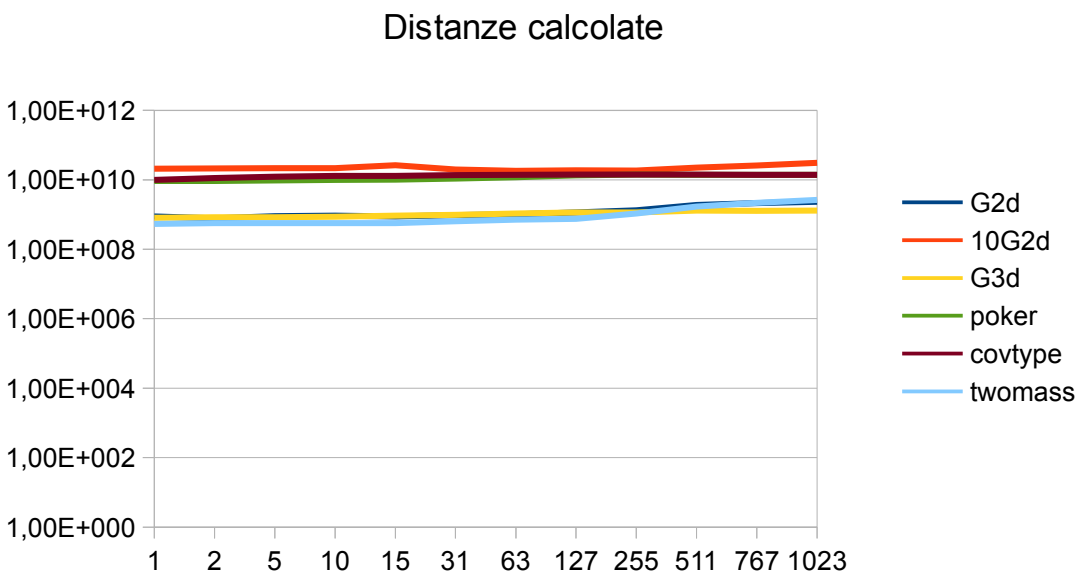


Inoltre, il numero delle iterazioni è direttamente proporzionale a n, perché più si aumenta n e più il peso dell'n-esimo outlier è simile al peso di un oggetto normale, e così discriminare gli outlier dagli oggetti normali diventa un compito sempre più difficile. Al variare di k invece, il numero di iterazioni non è predicibile e in generale dipende dalla distribuzione dei dati, mentre all'aumentare di m il numero di iterazioni diminuisce.

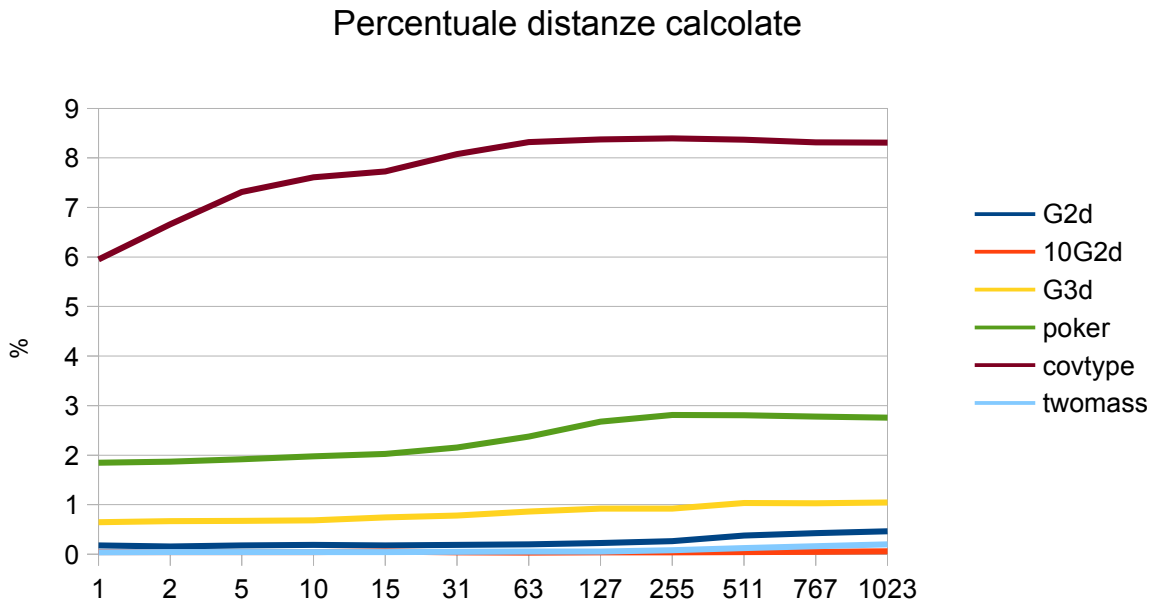
Dataset	$k = 50, m = 100$			$n = 10, m = 100$			$n = 10, k = 50$		
	$n = 10$	$n = 50$	$n = 100$	$k = 25$	$k = 50$	$k = 75$	$m = 50$	$m = 100$	$m = 200$
G2d $l = 255$	19	35	48	21	19	19	23	19	13
10G2d $l = 511$	34	50	69	36	34	34	41	34	25
G3d $l = 127$	34	51	59	29	34	33	49	34	20
Poker $l = 255$	142	206	236	129	142	154	280	142	73
Covtype $l = 255$	249	337	392	201	249	239	483	249	130
2Mass $l = 255$	9	44	72	37	9	7	14	9	7

Distanze calcolate

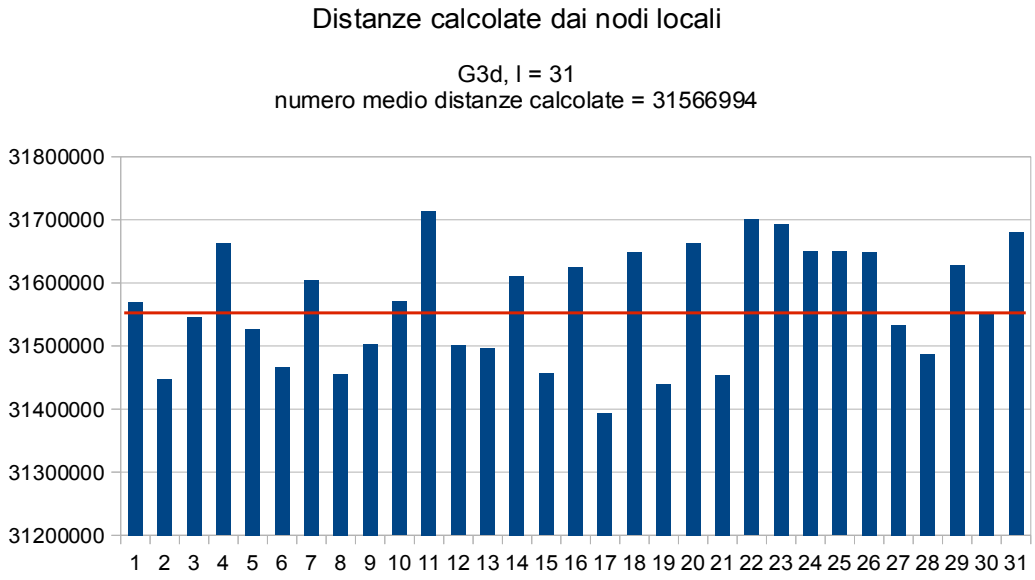
Il grafico sottostante riporta il numero di distanze calcolate al variare nei nodi locali impiegati. Possiamo notare che il numero di distanze calcolate varia lentamente con l'aumentare del numero dei nodi locali impiegati.



Qualunque sia il numero di nodi locali impiegati, il numero delle distanze calcolate è sempre solo una piccola percentuale del numero delle distanze che avrebbe calcolato un algoritmo *NaiveNestedLoop*. A parte i due dataset più impegnativi, *Covtype* e *Poker*, siamo sempre sotto l'1% di distanze calcolate.

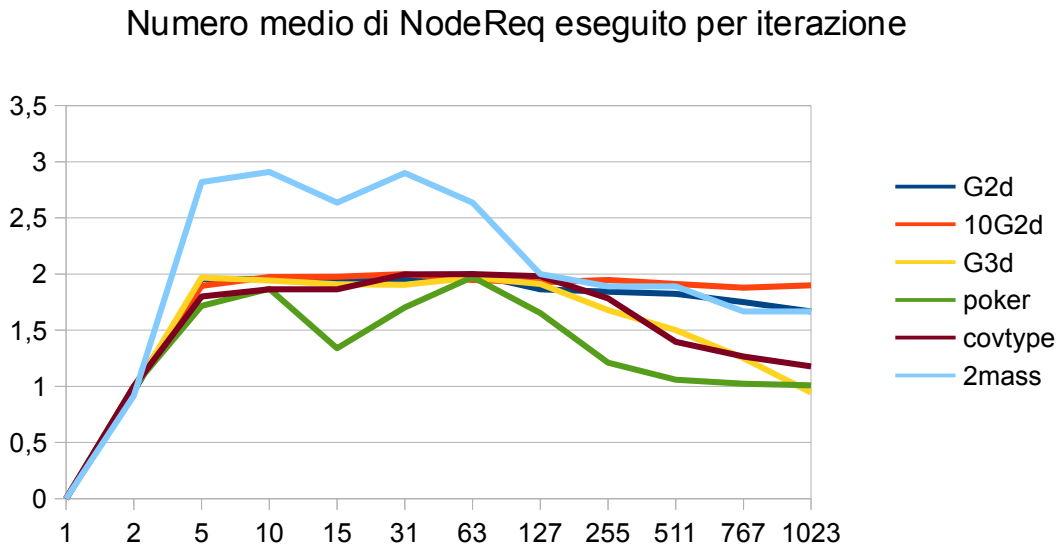


Il carico è inoltre ben bilanciato, ovvero il numero medio delle distanze calcolate dai nodi locali non discosta di molto dal numero effettivo di distanze calcolate dai diversi nodi. Sotto un esempio relativo al dataset *G3d* e $l = 31$. Il nodo 17 calcola un numero di distanze che è solo del 0,53% inferiore al valore medio.

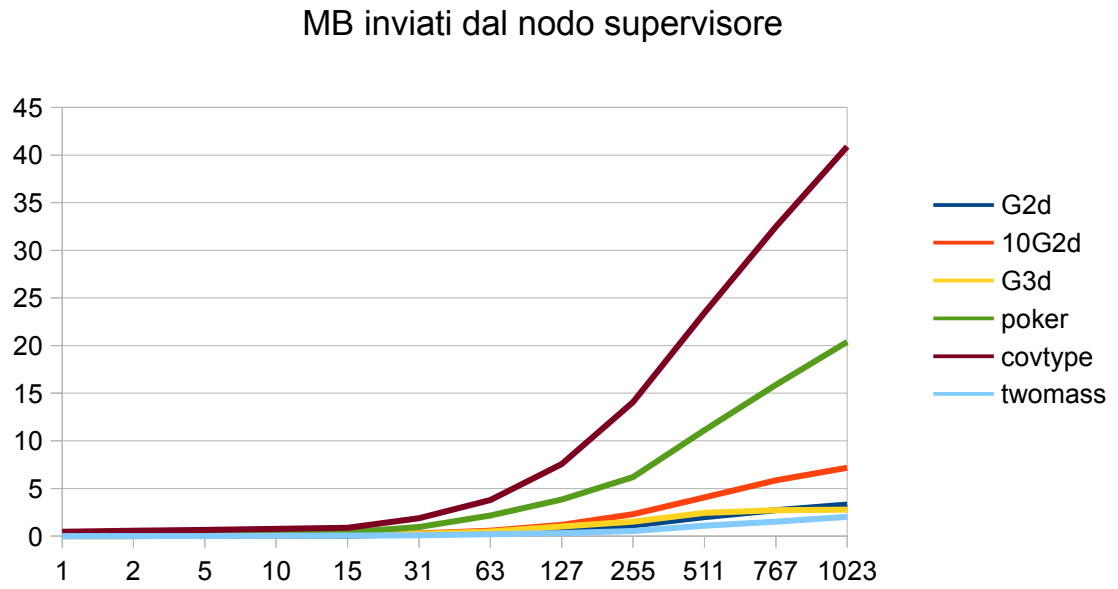


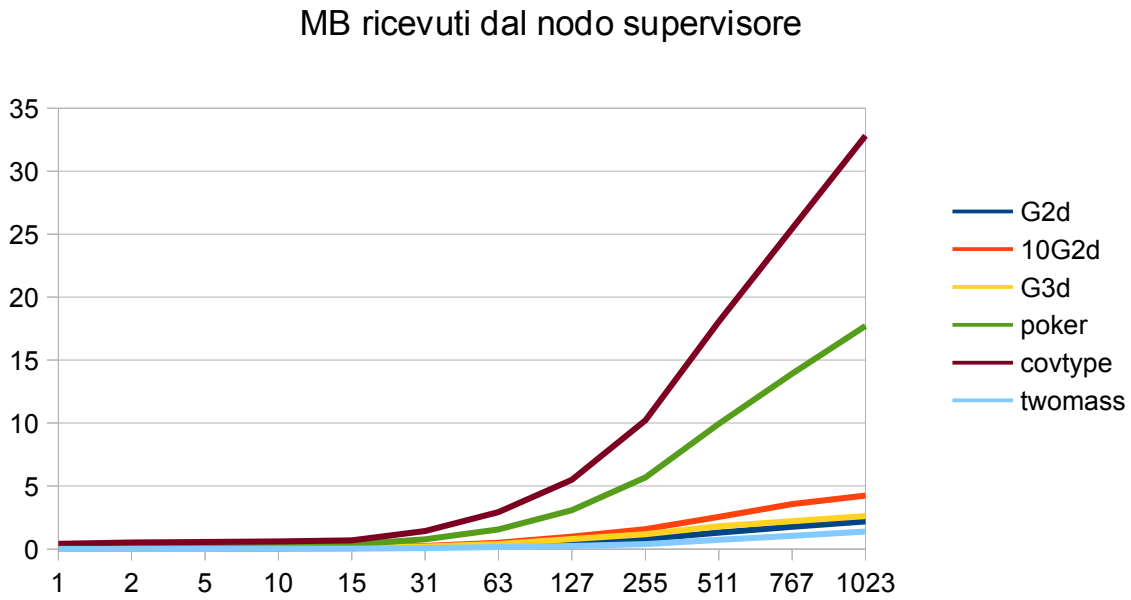
Quantità di dati trasferiti

Per limitare la quantità delle distanze distanze trasferite, l'algoritmo *LDSS* ha introdotto come effetto collaterale l'aumento di comunicazioni aggiuntive, in cui il nodo supervisore invia nuove informazioni ai nodi locali in modo da ricevere nuove distanze, tramite la procedura *NodeReq*. Il grafico sottostante mostra il numero medio di *NodeReq* eseguite per iterazione, e possiamo notare, fatta l'eccezione del dataset *2Mass*, come questo numero non vada mai oltre le due *NodeReq* per iterazione.



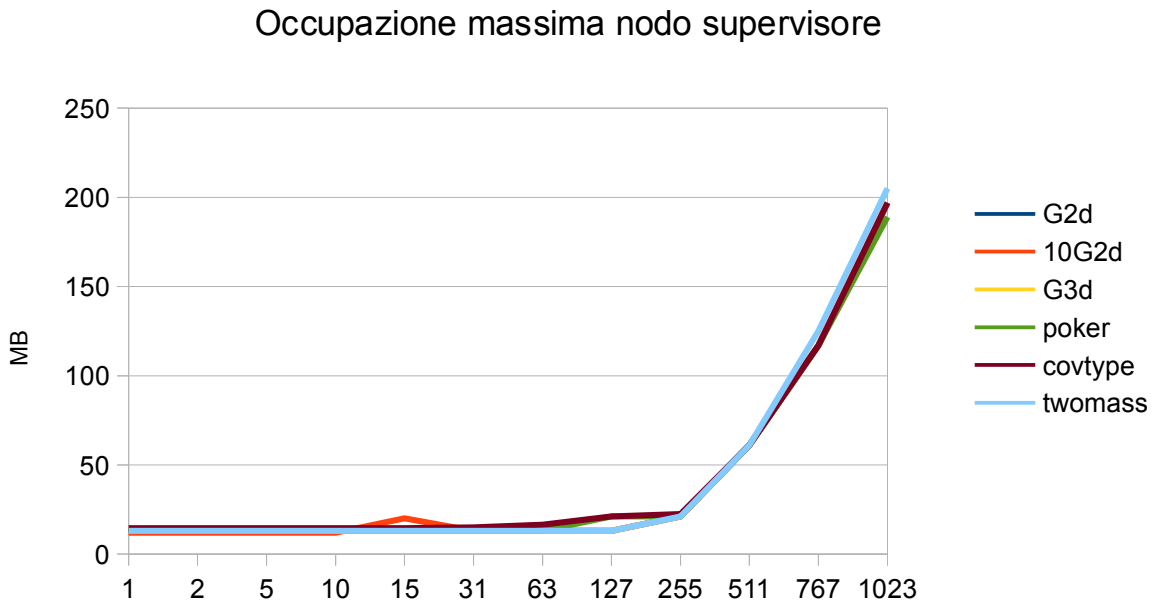
Possiamo notare inoltre che più i dataset sono "pesanti", come *CovType* e *Poker*, che richiedono un numero elevato di iterazioni, e più la quantità di dati trasmessi aumenta una volta superato il "limite" dei 63 nodi locali.

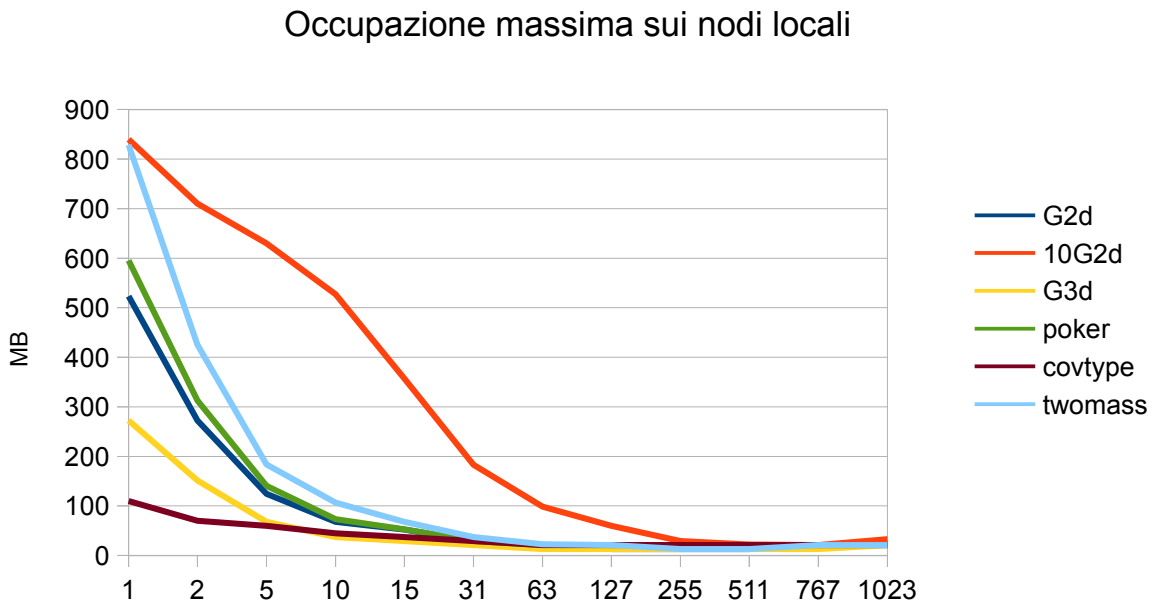




Occupazione di memoria

È stato tenuto conto anche dell'occupazione massima di memoria di ciascun processo. Come si può facilmente immaginare, con l'aumentare dei nodi locali diminuisce l'occupazione di memoria su di essi, dato che ciascun nodo sarà responsabile di un numero sempre inferiore di oggetti. Nel frattempo però aumenta l'occupazione di memoria sul nodo supervisore. I due grafici mostrano l'occupazione massima rilevata sul nodo supervisore e sui nodi locali.





Controllando i report degli esperimenti si è potuto constatare come l'occupazione di memoria massima sia praticamente uguale su ciascun nodo locale.

8 Conclusioni

L'obiettivo posto all'inizio di questa tesi era quello di implementare l'algoritmo Lazy-DistributedSolvingset, visto nel capitolo 4, e eseguire dei test al fine di verificare il suo comportamento quando eseguito in un ambiente ad alte prestazioni. I test sono stati eseguiti su FERMI, un IBM Blue Gene/Q disponibile presso il centro di calcolo del CINECA. Grazie a questa opportunità non abbiamo avuto limiti sulla scelta del numero di processi da allocare in ciascun test.

Il lavoro svolto è consistito in una prima fase di studio in cui ho analizzato l'algoritmo, mi sono documentato su MPI, l'interfaccia di scambio di messaggi che è stata utilizzata per la comunicazione tra i nodi, e ho preso familiarità col linguaggio C++. Questa è stata infatti la mia primissima esperienza con questo linguaggio e la maggior parte delle difficoltà iniziali che ho incontrato erano principalmente dovute alla mia inesperienza.

I test effettuati sono stati diversi. Fissata una configurazione di n, k e m sono stati eseguiti, per ciascuno dei sei dataset considerati, una serie di test partendo dall'utilizzo di un solo nodo locale fino ad arrivare all'impiego di 1023 nodi locali, per un totale di 12 test per dataset. Inoltre sono stati eseguiti dei test per osservare il comportamento al variare di questi tre parametri di configurazione.

I risultati sperimentali hanno confermato molti comportamenti attesi, tuttavia ci sono state anche sorprese non molto piacevoli. Come ci si aspettava, il numero volte che il nodo supervisore richiede nuove distanze ai nodi locali è relativamente basso, e tipicamente sta attorno alle due richieste per iterazione. Questa è un'ulteriore conferma del fatto che non è assolutamente necessario che ogni nodo locale invii tutte le sue prime k distanze per ciascun candidato, come invece avviene nell'algoritmo *DistributedSolvingSet*.

Abbiamo visto che il numero delle distanze calcolate varia molto lentamente, e che è solo una piccolissima percentuale rispetto al numero di distanze che verrebbero calcolate da un algoritmo *NaiveNestedLoop*. Inoltre, ogni nodo locale calcola un numero molto simile di distanze. Come previsto, anche la dimensione del solving set è sempre molto piccola e in genere è attorno al 1% della dimensione del dataset globale.

Al variare del numero dei nodi, il numero delle iterazioni necessarie per la risoluzione di uno specifico problema è circa costante, e comincia a diminuire solo quando si comincia a

impiegare un numero di nodi significativo. Fissati *k* e *m*, il numero di iterazioni è direttamente proporzionale al valore di *n*, mentre facendo variare *k* in generale non è possibile predire se il numero di iterazioni sarà maggiore o minore rispetto al caso da confrontare. Anche il parametro *m* incide sul numero di iterazioni: più è alto e più oggetti verranno presi in considerazione a ogni iterazione, e di conseguenza il numero di iterazioni necessarie è inversamente proporzionale al valore di questo parametro.

Come già detto, alcuni risultati non sono stati proprio soddisfacenti. L'occupazione massima di memoria sul nodo supervisore è destinata ad aumentare con il numero dei nodi locali. Questo può essere un primo indizio del fatto che pian piano il nodo supervisore si ritrova un carico di lavoro sempre più importante, il che è confermato osservando l'andamento del rapporto tra il tempo di esecuzione sul nodo supervisore e il tempo di esecuzione totale: con un numero di nodi alto (oltre i 127), il tempo di esecuzione passato sul nodo coordinatore diventa significativo. L'alto rapporto però è dovuto anche al fatto che con molti nodi il carico sui nodi locali è minimo. Questo può far venire il dubbio che forse abbiamo esagerato un po' troppo con il numero dei nodi impiegati.

Oltre a questo, anche il tempo di comunicazione tra i processi è destinato a crescere e ad avere un impatto negativo sulle performance. È stato evidenziato come in certe configurazioni il tempo di comunicazione arriva a toccare anche il 50% del tempo di esecuzione totale.

Questi risultati portano a pensare che le stime dei costi dell'algoritmo non siano del tutto indipendenti dal parametro *l*. I risultati ottenuti affermano però che questo parametro può essere trascurato quando impieghiamo un numero di nodi relativamente basso.

Anche se in certe situazioni lo speedup registrato con un numero “basso” di nodi può essere incoraggiante (ad esempio, per il dataset 2*Mass* abbiamo uno speedup di 100x con 127 nodi), personalmente penso che non ha comunque molto senso eseguire tali lavori su FERMI. Sistemi come FERMI sono pensati per eseguire codice che scali pesantemente, fino almeno 1024 processi. Inoltre un suo singolo core non è “velocissimo” (1.6 GHz), e di conseguenza i risultati ottenuti con pochi nodi potrebbero essere anche meno soddisfacenti di quelli ottenuti su dei sistemi più potenti (dal punto di vista del singolo core), e più facilmente accessibili.

Riguardo gli sviluppi futuri, si dovrebbe innanzitutto individuare la causa dell'alto tempo di comunicazione per cercare di limitarlo il più possibile, ma bisognerebbe tenere in considerazione anche il problema del carico sul nodo supervisore: non sarebbe male eliminare questi tempi di esecuzione, magari tramite una strategia non centralizzata che prevede una comunicazione tra i nodi in stile peer-to-peer.

9 Bibliografia

1. F. Angiulli, S. Basta, S. Lodi e C. Sartori, “Distributed Strategies for Mining Outliers in Large Data Sets”, IEEE Transactions on Knowledge and Data Engineering, vol. 25 n. 7, 2013.
2. Pagina ufficiale del MPI forum: http://www.mpi-forum.org/
3. Pagina ufficiale di MPICH: http://www.mpich.org/
4. Portale HPC del CINECA: http://www.hpc.cineca.it/
5. Tesi: “Implementazione di algoritmi di data mining in architetture a elevato parallelismo” di Matteo Zanirati.
6. Tesi: “Scoperta di outliers: implementazione e ottimizzazione di algoritmi su architetture distribuite” di Emma Coradduzza.