



SAPIENZA  
UNIVERSITÀ DI ROMA

## Sviluppo e sperimentazione di algoritmi su grafi in Pregel e Mapreduce

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica  
Corso di Laurea Magistrale in Informatica

Candidato

Luigi Piccioli  
Matricola 1434713

Relatore

Prof. Irene Finocchi

Anno Accademico 2014/2015

Tesi non ancora discussa

---

**Sviluppo e sperimentazione di algoritmi su grafi in Pregel e Mapreduce**  
Tesi di Laurea Magistrale. Sapienza – Università di Roma

© 2015 Luigi Piccioli. Tutti i diritti riservati

Questa tesi è stata composta con L<sup>A</sup>T<sub>E</sub>X e la classe Sapthesis.

Versione: 19 settembre 2015

Email dell'autore: luigipiccioli@gmail.com

# Indice



# Capitolo 1

## Introduzione



# Capitolo 2

## MapReduce

*MapReduce* [?] è un modello di programmazione in grado di operare su grandi quantità di dati, dove la computazione è automaticamente distribuita, parallelizzata su un cluster di computer. Oltre al modello di programmazione, *MapReduce* definisce il modello dell’architettura del cluster in cui viene eseguito il programma. In questo modo vengono mantenute separate ed indipendenti la componente implementativa di un algoritmo in *MapReduce*, che si limita alla dichiarazione delle sue funzioni base **MAP** e **REDUCE**, dal framework che si occupa della gestione delle risorse il cluster, del controllo dell’esecuzione del flusso del programma, del controllo degli errori e del recupero successivo al guasto di una o più macchine del cluster.

### 2.1 MAP e REDUCE

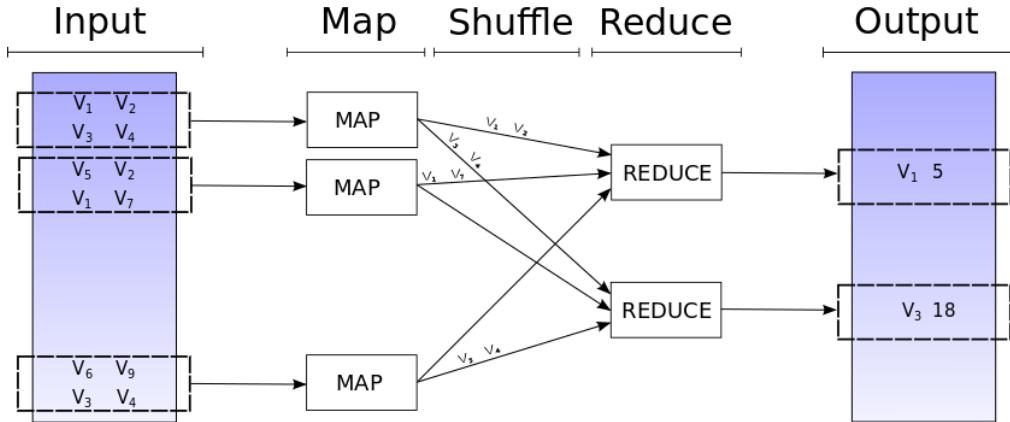
Un algoritmo in MapReduce [?] è definito attraverso le due funzioni base del modello, la funzione **MAP** e la funzione **REDUCE**. L’input viene espresso in coppie di valori  $\langle K, V \rangle$  con  $K$  chiave e  $V$  il valore associato a  $K$ . Le 2 funzioni base sono definite secondo le seguenti funzioni:

- **MAP**  $\langle K_{in}, V_{in} \rangle \Rightarrow list(\langle K_{out}, V_{out} \rangle)$
- **REDUCE**  $\langle K_{in}, list(V_{in}) \rangle \Rightarrow list(V_{out})$

La funzione di **MAP** prende in input la coppia di valori  $\langle K_{in}, V_{in} \rangle$  e produce in output la lista di valori intermedi  $\langle K_{out}, V_{out} \rangle$ , la lista di valori associati alla chiave  $K_{out}$ , che è raggruppata in modo trasparente dal framework *MapReduce* con le liste associate a  $K_{out}$  prodotte dalle altre funzioni *Map* che operano sul cluster.

La funzione **REDUCE** riceve come input una coppia di valori  $\langle K_{in}, list(V_{in}) \rangle$  dove  $list(V_{in})$  è l’insieme di tutti valori associati alla chiave  $K_{in}$  presenti nell’intero input. La funzione **REDUCE** può restituire in output una lista di valori associati alla chiave  $K_{in}, list(V_{out})$ .

Tra le funzioni **MAP** e **REDUCE** opera, in modo trasparente, la funzione di **SHUFFLE** la quale raggruppa gli output delle funzioni **MAP**,  $list(\langle K_{out}, V_{out} \rangle)$ , che condividono la medesima chiave. Vengono così create le coppie di valori  $\langle K_{in}, list(V_{in}) \rangle$  che verranno prese in input dalle funzioni **REDUCE**.



**Figura 2.1.** Esempio MapReduce - Calcolo del grado dei nodi di un grafo orientato

### 2.1.1 Esempio MapReduce

Un esempio semplice di algoritmo in *MapReduce* [?] è il calcolo del grado dei nodi di un grafo orientato. Il grafo in input è rappresentato dalla lista di archi nel formato  $\langle V_a, V_b \rangle$  dove  $V_a$  e  $V_b$  rappresentano reciprocamente il vertice di partenza e il vertice di destinazione dell'arco in input.

La funzione **MAP** riceve in input la coppia  $\langle V_a, V_b \rangle$  e produce in output la coppia  $\langle V_a, V_b \rangle$ . In modo trasparente agisce la funzione **SHUFFLE** che raggruppa l'output di ogni *Mapper* in base al valore  $V_a$  e lo instrada verso la funzione **REDUCE**, questo garantisce che tutti gli archi di un vertice  $V_a$  vengano presi in input dalla stessa funzione **REDUCE**.

La funzione **REDUCE** riceve in input  $\langle V_a, [V_{b1}, V_{b2}, \dots, V_{bn}] \rangle$ , dove  $V_a$  è la chiave del vertice e  $[V_1, V_2, \dots, V_n]$  è l'insieme nodi vicini al nodo  $V_a$ . La funzione produce in output la coppia  $\langle V_a, \text{degree}(V_a) \rangle$  dove  $\text{degree}(V_a)$  è il grado del nodo  $V_a$  dato dalla cardinalità dell'insieme di nodi  $[V_{b1}, V_{b2}, \dots, V_{bn}]$ .

#### Pseudocode 2.1. Pseudocodice funzione MAP

```

map(String v, String u){
    // key: vertice di partenza arco
    // value: vertice destinazione
    emit(v, u);
}

```

**Pseudocode 2.2.** Pseudocodice funzione **REDUCE**

```

||| reduce(String v, Iterator vicini){
|||     // v: ID Vertice v
|||     // vicini: Lista vertici vicini
|||     int degree= 0;
|||     foreach u in vicini{
|||         degree += 1;
|||     }
|||     emit(v,degree);
|||
}

```

## 2.2 Architettura MapReduce

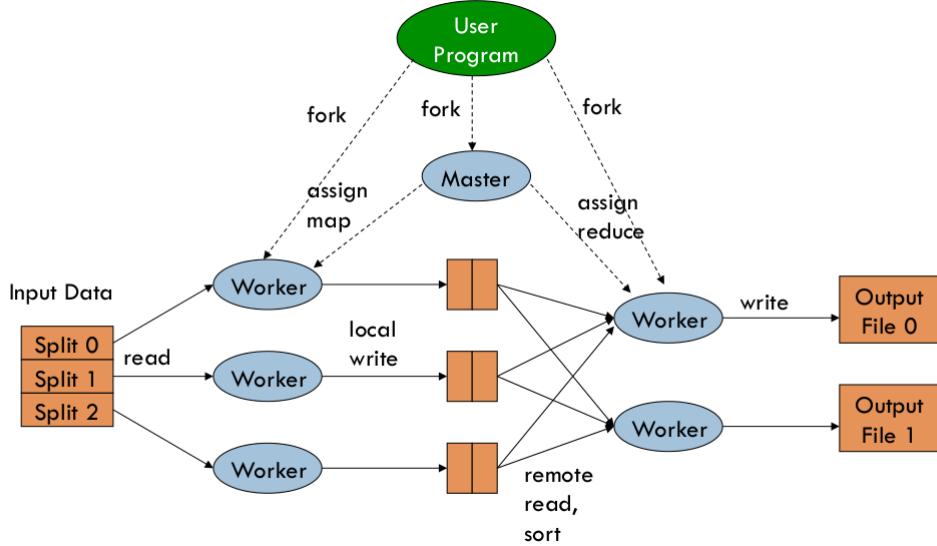
In questo paragrafo viene descritta l'architettura definita in *MapReduce* [?] per la gestione del flusso del programma sul cluster di computer.

Il modello dell'architettura *MapReduce* è composto da un nodo **Master** che si occupa della gestione del flusso dei dati e dell'assegnazione dei *task* a tutti i nodi del cluster, i **Worker**. Il *Master* può assegnare ai singoli *Worker*, funzioni di *MAP*, in questo caso il *Worker* viene definito *Mapper*, o funzioni *REDUCE*, in questo caso viene definito *Reducer*.

### 2.2.1 Flusso di un programma MapReduce

In Figura ?? viene rappresentato lo schema del flusso di programma eseguito su l'architettura *MapReduce*. Quando viene inviata un operazione *MapReduce* al cluster, il flusso del programma segue 7 passi:

1. L'input del programma viene suddiviso in N parti distinte aventi la stessa dimensione.
2. Le copie del programma vengono distribuite a tutte le macchine del cluster (sia Master che Worker)
3. Il Master controlla quali worker si trovano in uno stato INDLE è assegna loro i task Map e Reduce
4. I Worker che ricevono un task MAP caricano una delle N parti dell'input, leggono le coppie <Key,Value> e salvano nella memorie locali i risultati intermedi generati.
5. I risultati salvati localmente vengono ripartiti utilizzando la funzione di partizione.
6. Il nodo Master riceve la posizione dei risultati partizionati e la comunica ai Reducer a cui sono assegnate la partizioni
7. I Reducer caricano i dati di una partizione dalle memorie locali dei Mapper, una volta caricati tutti i dati relativi ad una partizione eseguono la funzione REDUCE e producono in output i risultati.



**Figura 2.2.** Architetture MapReduce - Flusso di un programma

Al termine dell'elaborazione, il programma *MapReduce* genera  $R$  file di output, dove  $R$  è il numero di *Reducer* che sono stati utilizzati.

### 2.2.2 Funzione di Partizione

Per assicurare che ogni *Reducer* ottenga tutti i risultati parziali relativi ad una chiave, prodotti dai *Mapper*, viene utilizzata una funzione di partizione unica. In questo modo lo spazio delle chiavi è ripartito in modo uniforme sia dal *Master* che dai *Worker* del cluster. Un esempio di funzione di partizione di base utilizzata in *MapReduce* [?] è:

$$\text{hash}(\text{key})/\text{mod } R$$

dove  $R$  è il numero di partizioni, che solitamente è uguale al numero di *Reducer*.

Nel esempio precedente, dove è calcolato il grado dei nodi di un grafo orientato, applicare la funzione di partizionamento  $\text{hash}(V_a) \text{ mod } R$  assicura che tutti le coppie di valori  $\langle V_a, V_b \rangle$  con chiave  $V_a$  vengano assegnate, da tutti i *Worker*, alla stessa partizione.

### 2.2.3 Master

Il nodo *Master* del cluster si occupa di assegnare e controllare lo stato del progresso dei **Task**, Map o Reduce, da far svolgere ai *Worker*.

Gli stati in cui possono trovarsi i task sono:

1. **Idle**, il task non è stato ancora assegnato a nessun worker
2. **In-progress**, il task è stato assegnato ed è in esecuzione presso il worker associato
3. **Completed**, il nodo ha terminato il task assegnato.

Inizialmente tutti i task che compongono il programma MapReduce si trovano in uno stato **Idle**. Il nodo *Master* verifica la disponibilità delle risorse sui vari *Worker* ed assegna i Task, che passano allo stato **In-progress**

Appena un task **Map** passa ad uno stato **Completed**, il *Master* viene informato della posizione dei risultati parziali che sono stati generati dal *Mapper*. I risultati parziali vengono salvati momentaneamente all'interno della memoria locale del *Mapper*. Il nodo *Master* notifica la posizione di memoria ricevuta, appena possibile, al *Reducer* a cui è associata la partizione. I processi **Reduce** iniziano ad eseguire la funzione *Reduce* non appena tutti i task *Map* passano ad uno stato *Completed*, significando che l'intero input è stato letto e partizionato. Una volta che lo stato di tutti i task *Reduce* passano allo stato *Completed*, l'output del programma è stato salvato su file system distribuito ed il *Master* termina con successo la computazione.

#### 2.2.4 Controllo e gestione degli errori

MapReduce [?] per essere in grado di parallelizzare il processo su cluster composti anche da migliaia di macchine, è essenziale la fase di controllo costante del corretto funzionamento di ogni componente cluster coinvolto nel processo.

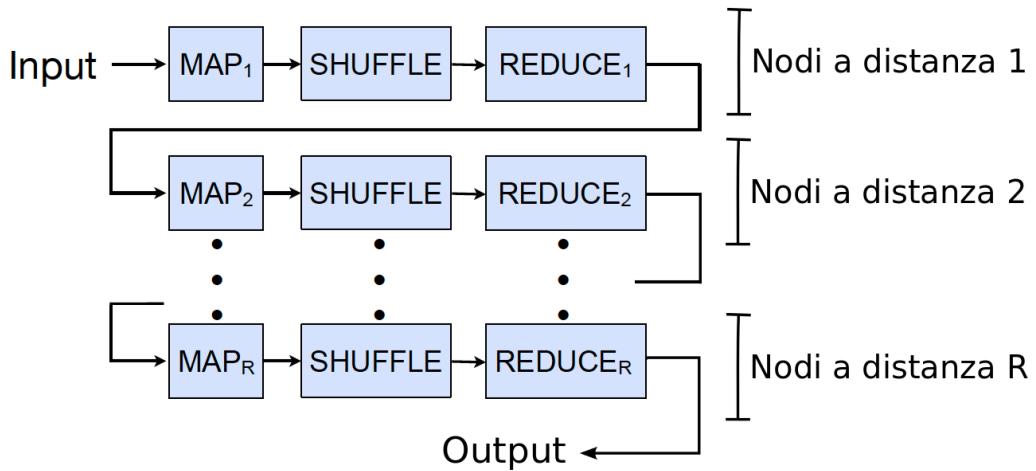
Il nodo *Master* si occupa di verificare che i *Worker* siano attivi attraverso l'invio di ping ad intervalli regolari. Nel caso in cui il *Master* non riceva risposta da uno dei nodi *Worker*, il nodo viene marchiato come **fallito**. Il fallimento di un nodo causa la riassegnazione e la rielaborazione di alcuni task assegnati al nodo fallito, in modo più dettagliato:

- I task *Map* completati nel momento in cui un nodo risulta essere fallito, vengono riportati allo stato iniziale *idle*, in attesa di essere riassegnati ad un altro *Mapper*. La rielaborazione di un task *Map* già completato è necessaria in quanto il fallimento della macchina potrebbe rendere inaccessibile i risultati parziali ai *Reducer*.
- I task *Map* e *Reduce* che si trovano in uno stato *In-progress* vengono riportati allo stato iniziale *idle*, in attesa di essere riassegnati ad un altro *Mapper* o ad un altro *Reducer*.
- I task *Reduce* che hanno raggiunto lo stato *Completed* non sono influenzati dal fallimento del nodo a cui erano assegnati in quanto il loro output non è salvato in locale ma su file system distribuito.

Il fallimento del nodo *Master* causa il fallimento del programma *MapReduce* e la perdita di tutti i risultati parziali calcolati fino a quel momento. La probabilità che sia il nodo *Master* a fallire è comunque molto bassa rispetto alla probabilità di fallimento di uno dei nodi *Worker*, che possono arrivare a contare anche migliaia di unità.

## 2.3 MapReduce - Iterazioni a cascata

Non tutti gli algoritmi sono implementabili nel modello MapReduce [?] utilizzando una singola iterazione del programma. Per implementare alcuni algoritmi è necessario



**Figura 2.3.** MapReduce - Iterazioni a cascata

eseguire più round MapReduce successivi, ad ogni passaggio l'output ottenuto in un round diventa l'input per il round successivo.

In [?] viene descritta la classe di complessità degli algoritmi *MapReduce*,  $MRC^i$ . Gli algoritmi che appartengono alla classe  $MRC^i$  devono rispettare una serie di vincoli che limitano la complessità delle funzioni Map e Reduce e del numero di macchine utilizzate, rispetto alla dimensioni dell'input e viene definita in rapporto al numero di round  $R$  eseguiti, con  $R = O(\log^i n)$ .

Nell'esempio visto precedentemente, il calcolo dei nodi di un grafo orientato, viene eseguito in un solo round *MapReduce*. Il numero di iterazioni totali è costante ed uguale a  $1 = \log^0 n$ , l'algoritmo appartiene quindi alla classe  $MRC^0$ .

Il numero di iterazioni è un fattore importante ed influenza le complessità dell'algoritmo *MapReduce* in quanto al crescere del numero di round aumentano il numero di operazione *Shuffle*, la quale risulta un'operazione molto costosa.

**Shuffle**, cioè la funzione che viene svolta in modo trasparente dal modello MapReduce [?], ordina l'output di ogni *Mapper*, lo ripartisce e lo instrada verso il corretto *Reducer*. In questa fase vengono notificate le posizioni dei risultati parziali al nodo *Master*, in questo modo viene introdotto un collo di bottiglia che rende costosa la fase di *Shuffle*. Questo comporta che all'aumentare delle iterazioni *MapReduce* si ha un aumento della complessità del programma. Inoltre ad ogni iterazione si ha un ripetersi delle operazioni di lettura e scrittura dei dati dal file system distribuito, che andranno ad incidere negativamente sulle prestazioni totali.

Un esempio di algoritmo su grafi, che richiede un implementazione *MapReduce* con più iterazioni, è il calcolo della distanza dei nodi di un grafo da un nodo sorgente, *Single Source Shortest Path (SSSP)*. Quest'ultimo verrà introdotto più dettagliatamente nel corso del 4° capitolo.

*SSSP*, ad ogni iterazione prende in input la lista di archi del grafo nel formato  $\langle a : \text{distanza}_a, b : \text{distanza}_b \rangle$ , dove il valore *distanza* è inizializzato, nel corso primo round, ad  $\infty$  per tutti i nodi a parte il nodo sorgente, inizializzato con valore *distanza* 0.

La funzione **Map** prende la coppia valori  $\langle a:distanza_a, b:distanza_b \rangle$  e produce in output la coppia  $\langle a:distanza_a, b:distanza_b \rangle$ .

La funzione **Reducer** riceve la lista dei nodi vicini del nodo nel formato  $\langle a:distanza_a, [b_1:distanza_{b1}, b_2:distanza_{b2}, \dots, b_n:distanza_{bn}] \rangle$  e aggiorna i valori  $distanza_{bi}$  dei nodi vicini, con  $i = 1, \dots, n$  con il valore minore tra il valore  $distanza_{bi}$  vicino e il valore  $distanza_a + 1$ , più precisamente:

```
reduce(String key, Iterator values){
    // key: ID Vertice
    // values: Lista vertici vicini
    int dist_min = key.distanza;
    for each v in values{
        if(dist_min > v.distanza + 1 ){
            dist_min = v.distanza + 1;
        }
    }
    for each v in values{
        emit(v,k:dist_min+1)
    }
}
```

Le iterazione MapReduce terminano quando l'algoritmo converge e non avvengono più modifiche ai valori distanza.

## 2.4 Implementazioni di MapReduce

Esistono numerose framework che implementano e/o includono il modello MapReduce [?], tra i più utilizzati troviamo:

- Apache Hadoop [?]
- Google MapReduce [?]
- MongoDB [?]
- Aster Data [?]

Per l'implementazione di algoritmi su grafi in MapReduce, ho deciso di utilizzare Apache Hadoop [?], essendo la soluzione Open Source più utilizzata e meglio documentata.

Inoltre permette un confronto più diretto degli algoritmi sviluppati su Apache Giraph [?], un framework che implementa il modello Pregel [?], sviluppato integrandolo su Apache Hadoop e che verrà approfondito nel prossimo capitolo.

Le componenti principali, vedi ?? , di Apache Hadoop sono :

- **Hadoop Distributed File System (HDFS)**: File system distribuito
- **Hadoop YARN, Yet Another Resource Negotiator**: A framework per la gestione delle risorse del cluster.
- **Hadoop MapReduce**: Implementazione del modello MapReduce



**Figura 2.4.** Apache Hadoop

Ogni programma implementato in *Apache Hadoop* istanzia uno o più **Job**, cioè programmi che seguono il modello MapReduce. La definizione di un *Job* è suddiviso in 2 parti, la configurazione del cluster e l'implementazione delle funzioni di *Map* e di *Reduce*. La configurazione del cluster su cui verrà eseguito il Job prevede, per esempio, la definizione del numero di *Mapper* e *Reducer* che verranno utilizzati. **Hadoop MapReduce** si occupa della gestione del flusso del Job, lo divide in sotto-parti, i task, che verranno assegnati ai Worker presenti cluster, i Container. Le risorse del Cluster vengono gestite da **Hadoop YARN** che istanzia, a seconda delle disponibilità delle risorse sulle singole macchine, i *Container*. I container sono le entità che andranno a eseguire i *task* assegnati. **Hadoop Distributed File System** è l'implementazione di file system distribuito associato alla distribuzione Apache Hadoop, utilizzato dai *Mapper* per caricare i dati dell'input e dai *Reducer* per scrivere l'output.

## Capitolo 3

# Pregel

Il modello **Pregel** [?] è un modello per il calcolo distribuito, parallelo e fault-tollerance di algoritmi su grafi su cluster di computer. *Pregel* può essere definito come una specializzazione di *MapReduce* [?] sui grafi. Come avviene in *MapReduce* il modello è composto sia dalla specifica dell'architettura su cui eseguire il programma sia dalla specifica del modello di programmazione, in modo da mantenerle separate ed indipendenti le due parti.

Un programma sviluppato in *Pregel* permette di definire algoritmi su grafi in modo più intuitivo rispetto a *MapReduce*, inoltre permette di evitare i costi delle numerose operazioni di I/O e delle operazioni di Shuffle, viste nel capitolo precedente. Questi costi sono evitati perché non è più presente l'inefficienza causata dalla presenza di molteplici iterazioni dell'algoritmo, iterazioni spesso necessarie per l'implementazione di algoritmi su Grafi, come nell'esempio di SSSP visto nel capitolo precedente.

Un programma che segue il modello Pregel, è espresso dal punto di vista del vertice di un grafo, ad ogni iterazione  $S$ , chiamata Superstep con  $S = 0, 1, 2, \dots$ , un vertice in Pregel:

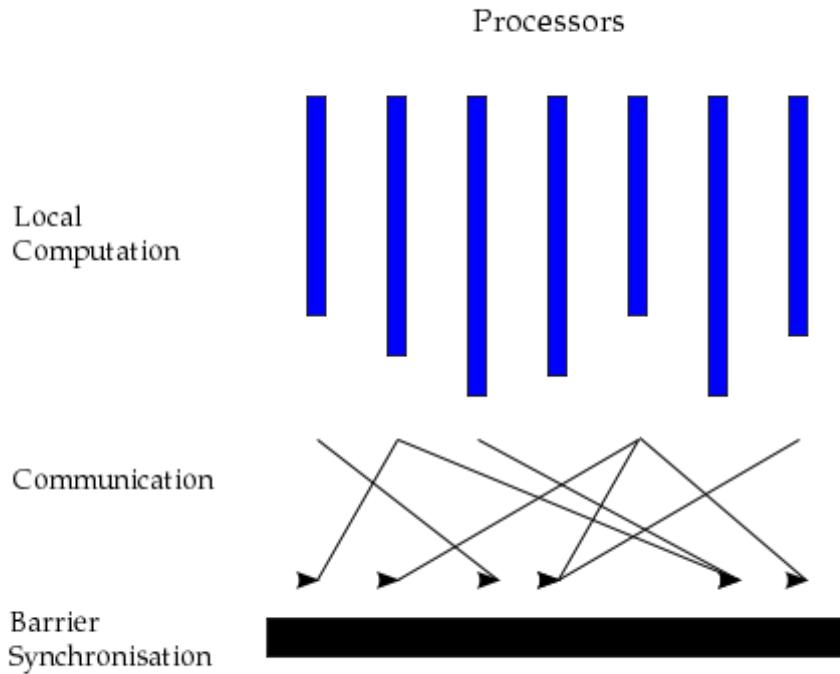
- Riceve i messaggi inviati durante Superstep  $S-1$ , l'iterazione Pregel precedente.
- Invia i messaggi ad altri vertici del grafo, che verranno ricevuti nel Superstep successivo  $S+1$ .
- Modifica il suo stato e/o lo stato dei suoi archi uscenti.

### 3.1 Bulk Synchronous Parallel (BSP)

*Pregel* si basa sul modello descritto in [?], il **Bulk Synchronous Parallel (BSP)**. *BSP* è un modello computazionale astratto per la progettazione di algoritmi paralleli. Una macchina *BSP* è costituita da più processori, ognuno avente la propria memoria locale indipendente e connessi tra loro da una rete di comunicazione.

Una computazione definita secondo *BSP* è composta da una serie di Superstep globali, ogni Superstep è composto da:

- **Calcolo concorrente:** Ogni processore che partecipa al Superstep effettua le operazioni utilizzando la propria memoria locale.

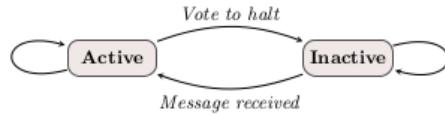


**Figura 3.1.** Bulk Synchronous Parallel (BSP)

- **Comunicazione:** In questa fase i processori scambiano dati tra loro.
- **Barriera di sincronizzazione:** Un processo che raggiunge questo step, rimane in stato di attesa fino a quando tutti i processi non hanno terminato la fasi precedenti.

La *Barriera di sincronizzazione* evita che si creino dipendenze circolari, rendendo impossibile il deadlock dei processi in *BSP*, inoltre permette l'implementazione di Checkpoint per la gestione di errori dovuti a fallimenti di una o più macchine che compongono il cluster. L'utilizzo della *Barriera di Sincronizzazione* può portare a conseguenze potenzialmente molto costose, basti a pensare ad un processo che richiede un tempo maggiore rispetto a gli altri processi e provoca la permanenza prolungata di quest'ultimi nello stato di attesa. Per evitare questo fenomeno è molto importante che ogni processore esegua le stesse funzioni su quantità di dati simili.

La computazione di un programma definito in **Pregel** [?] è suddivisa in Superstep. Ad ogni *Supertep* il modello prevede l'esecuzione di una funzione *Compute()* che può prevedere l'invio e ricezione di messaggi da e verso altri vertici del grafo. Un Superstep termina quando tutti i vertici del grafo hanno terminato la funzione *Compute()*.



**Figura 3.2.** Macchina degli stati di un Vertice in Pregel

## 3.2 Pregel

Il modello **Pregel** viene definito come un modello di computazione *vertice-centrico* su grafi diretti. La struttura di un vertice  $V$  in *Pregel* è composta da:

- **ID vertice:** Una stringa che identifica univocamente il vertice  $V$  sul grafo.
- **Stato del vertice:** Un valore mutabile durante l'esecuzione associata al vertice  $V$ .
- **Lista di archi uscenti:** L'insieme degli archi diretti che hanno il vertice  $V$  come nodo sorgente.

Gli archi in *Pregel* sono entità secondarie e sono associate alla struttura del vertice sorgente dell'arco. La struttura di un arco è composto da un valore mutabile durante l'esecuzione del Superstep e dall'identificativo del nodo di destinazione dell'arco.

Un programma in *Pregel* è costituito da 3 fasi principali:

- Nella prima fase viene caricato lo stato iniziale del grafo dell'input, per esempio da un file system distribuito.
- Nella seconda fase vengono eseguiti una serie di Superstep necessari all'esecuzione del algoritmo. Durante un Superstep, i vertici eseguono la stessa funzione sulla struttura del vertice e che definisce la logica dell'algoritmo.
- Nell'ultima fase viene scritto l'output del programma.

La terminazione dell'algoritmo è raggiunta quando tutti i vertici si trovano in uno stato *Vote to Halt*. Nel primo Superstep, tutti i vertici si trovano nello stato *Active*, ad ogni Superstep il vertice può passare o meno allo stato *Vote to Halt*.

Un vertice  $V$  che passa allo *Vote to Halt* nel Superstep  $s$ , rimane inattivo, cioè non effettua operazioni nei Superstep successivi a  $s$ .  $V$  rimane inattivo finché tutti i vertici passano allo stato *Vote to Halt* e viene terminata l'esecuzione del programma oppure, il vertice  $V$ , riceve un messaggio in entrata al Superstep  $r$  (con  $r > s$ ), questo fa tornare lo stato del vertice  $V$  in *Active*, e quindi ha effettuare operazioni nel corso del Superstep  $r$ .

Nella figura?? è rappresentata la macchina degli stati di un vertice in Pregel.

### 3.2.1 Esempio Pregel

Un esempio base di algoritmo in *Pregel* [?] è il calcolo del grado uscente dei nodi di un grafo orientato. L'implementazione in *Pregel* prevede la definizione di una

funzione *Compute()* che implementa le azioni che svolge un vertice nel corso del *Superstep*. L'implementazione dell'algoritmo per il calcolo del nodo del grafo prevede 1 *Supertep* in cui ogni vertice conta il numero di grafi uscenti associati alla propria struttura, assegna al proprio stato il valore associato alla struttura del vertice e passa allo stato *Vote to Halt*. Tutti i vertici passano allo stato *Vote to Halt* dopo il primo *Superstep*, i vertici salvano il proprio stato su File System Distribuito e il programma termina.

#### Pseudocode 3.1. Esempio Pregel

```

    compute(Vertex vertex, Messages messages){
        degree = 0;
        for each u in vertex.getEdegis(){
            degree += 1;
        }
        vertex.value = degree;
        vertex.voteTohalt();
    }
}

```

### 3.3 Architettura Pregel

Come avviene per *MapReduce* [?], il modello Pregel definisce anche l'architettura del cluster su cui è eseguito il programma, mantenendola separata e indipendente dalla componente in cui viene implementato l'algoritmo.

L'architettura del cluster ha un organizzazione di tipo *Master - Worker* simile all'architettura descritta in *MapReduce*. I vertici caricati in input vengono suddivisi tramite una funzione di partizione. Ogni partizione del grafo viene assegnata ad un *Worker*.

La funzione di partizione di default definita in *Pregel* è la funzione Hash:  $\text{hash}(\text{VERTEXID}) \bmod N$ , con  $N$  numero di nodi *Worker* nel cluster

*Pregel* prevede la possibilità di definire una funzione di partizione ad hoc per migliorare la partizione del grafo in modo da distribuire in modo uniforme i costi computazionali su tutti i *Worker*, per esempio il bilanciamento del grado totale dei vertici presenti nelle partizioni assegnate ai *Worker*. Questo permette di poter bilanciare il carico computazionale sui *Worker* adattandolo all'algoritmo che da implementare in *Pregel*.

#### 3.3.1 Flusso di un programma Pregel

L'esecuzione di un programma Pregel sul cluster segue i seguenti passi:

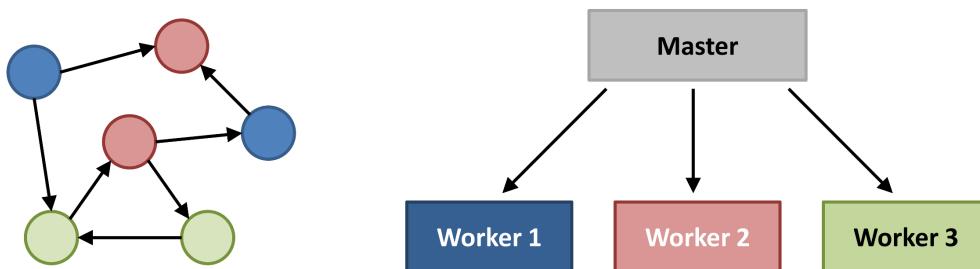
- Le copie del programma vengono distribuite a tutti i nodi del cluster, uno dei nodi viene eletto nodo *Master* con il compito di coordinamento dei nodi *Worker*, il nodo *Master* non ha partizione del grafo assegnate.
- Il nodo *Master* determina le partizioni del grafo e le assegna ad i *Worker*. I *Worker* si occupano di mantenere lo stato della partizione assegnata e di eseguire la funzione *Compute()* sui tutti i vertici che appartengono alla partizione.

- Il nodo *Master* assegna una porzione dell'input ad ogni *Worker*. I nodi *Worker* leggono la porzione di input assegnato e, nel caso in cui il vertice appartiene a una partizione assegnata allo stesso *Worker*, creano direttamente la struttura dati del vertice in memoria locale. Se il vertice in input, invece, appartiene ad una partizione assegnata ad un altro *Worker*, viene inviato un messaggio al *Worker* che creerà la struttura dati del vertice nella propria memoria locale. Finita la fase di caricamento del grafo, tutti i nodi del grado passano ad allo *Active*.
- Il *Master* comunica a tutti i *Worker* l'inizio del *Superstep*. I *Worker* eseguono la funzione *Compute()* sui vertici attivi della partizione assegnatagli, al termine i *Worker* inviano al *Master* il numero di nodi che sono nello stato *Active* nel *Superstep* successivo. Questo passaggio viene ripetuto finché sono presenti nodi *Active* nel grafo e sono presenti messaggi da recapitare nel *Superstep* successivo.
- Quando tutti i nodi si trovano nello stato *Vote to Halt* il *Master* comunica ai nodi *Worker* che la computazione è terminata e di salvare la propria partizione del grafo nell'output.

### 3.3.2 Master

Il nodo **Master** ha come funzione principale il coordinamento del lavoro dei nodi *Worker*. Il *Master* mantiene una struttura dati locale composta dal numero identificativo di ogni *Worker* attivo e le partizioni assegnate ad ogni *Worker*. Non viene mantenuta nessuna informazione riguardante il grafo in input, quindi mantenendo una struttura dati le cui dimensioni sono proporzionali al numero di partizioni del grafo e non al numero di vertici o archi del grafo stesso, permette di poter gestire il coordinamento di grafi di grandi dimensioni.

Le operazioni svolte dal *Master* seguono lo stesso ciclo di *Superstep* in cui viene suddivisa la computazione del *Worker*. Ogni volta che viene raggiunta una *Barriera di Sincronizzazione* il *Master* verifica, tramite ping, quali *Worker* sono ancora attivi. In caso non si verificano errori e tutti i *Worker* sono ancora attivi allora il *Master* fa avanzare la computazione al *Superstep* successivo. Se viene rilevato il fallimento di uno o più *Worker* si avvia la procedura di recupero che vedremo più avanti.



**Figura 3.3.** Partizionamento vertici in Pregel

Ad ogni ciclo il *Master* si occupa di mantenere una serie di informazioni statistiche sullo stato totale del grafo, come il numero di vertici e archi attivi oppure il numero di messaggi inviati, le informazioni sono raccolte nel corso di un *Superstep* e rese disponibili a tutti i *Worker* nel *Superstep* successivo.

### 3.3.3 Worker

I nodi **Worker** mantengono le strutture dati dei vertici che appartengono alla partizione del grafo assegnatagli durante la fase iniziale dal *Master*. La struttura dati di un vertice è composta dal suo identificativo, dal valore corrente del vertice, dalla lista di messaggi in entrata e dalla lista degli archi uscenti dal vertice; ogni arco è rappresentato dal indice del vertice di destinazione e dal valore associato all'arco stesso.

Il modello non prevede l'accesso alla lista degli archi in entrata in un nodo, archi che saranno gestiti dai vertici sorgente, Vertici che si potrebbero trovare o all'interno della partizione dello stesso *Worker* o su una partizione del grafo assegnata ad un *Worker* differente.

### 3.3.4 Aggregatori

In **Pregel** [?], gli aggregatori implementano il meccanismo per il calcolo di valori globali. Ogni vertice può inviare un valore al sistema di aggregazione, i valori inviati da tutti i vertici verranno aggregati e resi disponibili dal *Superstep* successivo. **Pregel** implementa una serie di aggregatori di esempio: MIN per il calcolo del valore minimo, MAX per il calcolo del valore massimo e SUM per la somma totale dei valori passati all'aggregatore. Un esempio di utilizzo dell'aggregatore MAX è calcolo del grado massimo dei vertici di un grafo orientato. Una volta ottenuto il valore del grado uscente del nodo, il valore viene inviato all'aggregatore. Alla fine del *Superstep* i *Worker* raccolgono i valori passati all'aggregatore MAX dai vertici ed ottengono un valore aggregato parziale. Ogni *Worker* invia al nodo *Master* il valore aggregato MAX parziale, il *Master* aggraga a sua volta i valori ottenendo il valore aggregato MAX globale, che rappresenta il grado massimo del vertice del grafo.

Alla fine del *Superstep*, il *Master* invierà il valore aggregato a tutti i *Worker* che sarà disponibile quindi dal *Superstep* successivo.

### 3.3.5 Gestione e recupero degli errori

Come in *MapReduce*, Pregel è in grado di parallelizzare il processo su cluster composti anche da migliaia di macchine, questo richiede un controllo dello stato di ogni macchina in modo da garantire il funzionamento dell'intera architettura. Il *Master*, come visto, controlla lo stato dei *Worker* tramite un sistema di Ping inviati periodicamente. Per garantire il recupero dello stato del grafo a seguito di un errore da parte di uno dei *Worker*, all'inizio di alcuni *Superstep*, il *Master* comunica ai *Worker* di salvare sul File System distribuito lo stato della propria partizione. Per non appesantire troppo il processo questa operazione non viene fatta all'inizio di ogni *Superstep* ma a determinati checkpoint stabiliti dal nodo *Master*.

Quando il *Master* rileva il fallimento da parte di uno o più *Worker* e di conseguenza la perdita dello stato corrente di alcune partizioni del grafo, procede alla

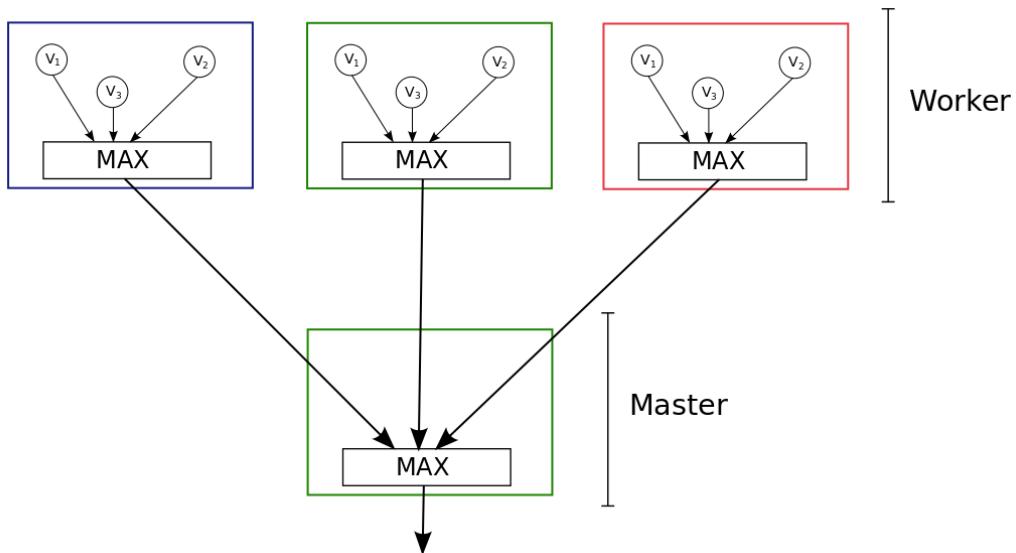


Figura 3.4. Aggregatore MAX

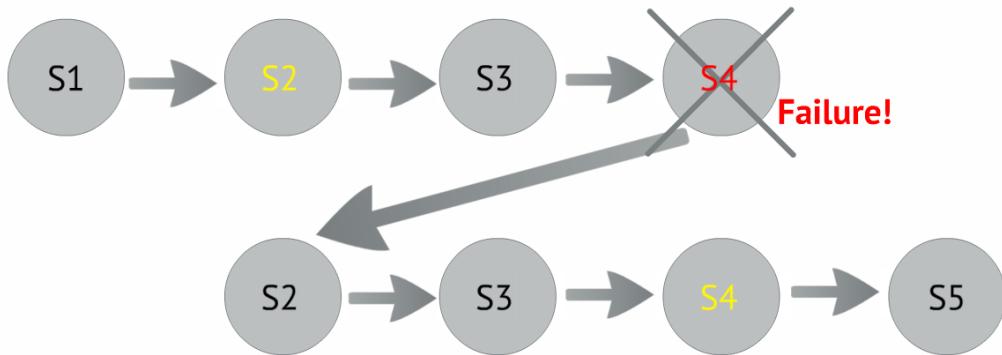


Figura 3.5. Checkpoint Pregel

riassegnazione delle partizioni appartenenti alle macchine fallite sui nodi *Worker* rimasti attivi. I *Worker* che ricevono nuove partizioni da gestire, caricano l'ultimo stato della partizione salvate sul file system distribuito. Il calcolo viene fatto ripartire dal *Superstep* successivo al checkpoint ripristinato, che si riferisce al *Superstep* successivo all'ultimo salvataggio dello stato del grado.

### 3.4 Giraph - Numero di iterazioni

Algoritmi su grafi che richiedono un numero elevato di iterazioni vengono implementati agevolmente nel modello Pregel, dove ad ogni Superstep viene applicata la logica delle iterazioni richieste. Al contrario di come avviene in *MapReduce*, il crescere del numero non aumenta la complessità del algoritmo, anzi la progettazione di algoritmi in *Pregel* è basata su azioni eseguite in una serie di iterazioni sul grafo.

Un esempio con più iterazioni è l'implementazione *Pregel* di *SSSP*, introdotto nel capitolo precedente e che verrà descritto più dettagliatamente nel capitolo successivo. In *SSSP* il valore associato ad ogni vertice è inizializzato ad  $\infty$  per tutti i nodi tranne per il nodo sorgente, inizializzato con valore 0, questi valori rappresentano la distanza minima dei nodi dal nodo sorgente.

La logica da applicare al singolo vertice nel corso di un *Superstep* è:

#### Pseudocode 3.2. Esempio Pregel

```

compute(Vertex vertex, Messages messages){
    dist_min = vertex.value;
    for each msg in messages{
        if(msg + 1 < dist_min):
            dist_min= msg + 1;
    }
    if(dist_min == vertex.value){
        //Valore distanza non aggiornato
        vertex.vote_to_halt();
    }
    else{
        //Valore distanza aggiornato
        vertex.value = dist_min;
    }
}

```

Nel *Superstep S* il vertice  $V$  riceve il valore *distanza minima* tramite un messaggio dai vertici a cui si trova vicino. Confronta le distanze ricevute con il proprio valore *distanza minima* attuale e lo aggiorna nel caso un nodo a cui si trova vicino sia a distanza minore dal nodo sorgente. Quando tutti i vertici, al termine di un *Superstep*, si trovano nello stato *Vote to Halt*, significa che nessun vertice ha aggiornato il proprio valore *distanza minima* nel corso dell'ultimo *Superstep* e il programma termina.

### 3.5 Implementazioni Pregel

Sono stati sviluppati diversi framework che implementano il modello Pregel [?], tra cui:

- Apache Giraph [?]
- GPS[?]
- Google Pregel [?]
- GraphLab [?]

in [?] sono stati confrontati tra loro i framework, analizzando le prestazioni dei vari sistemi su diversi algoritmi su grafi, come SSSP e PageRank, applicandoli su grafi di diverse dimensioni. Nel lavoro viene mostrato come i due sistemi, Apache Giraph e GPS, ottengono prestazioni migliori rispetto agli altri sistemi confrontati. Tra questi due framework si è deciso di utilizzare per Apache Giraph che è la soluzione OpenSource più documentata, meglio sviluppata e largamente supportata della comunità.



**Figura 3.6.** Apache Giraph

Un altro fattore che ha inciso sulla decisione di scegliere Apache Giraph come sistema di testing per il modello Pregel è il fatto che il framework è sviluppato integrandolo direttamente sulla infrastruttura di Apache Hadoop, questo ha reso possibile effettuare un confronto più diretto e preciso delle qualità prestazionali tra due sistemi.



## Capitolo 4

# Algoritmi implementati

Il confronto tra i Modelli *MapReduce* [?] e *Pregel* [?] è stato effettuato mettendo a confronto le loro implementazioni più utilizzate, rispettivamente *Apache Hadoop* [?] per quanto riguarda *MapReduce* e *Apache Giraph* [?] per quanto riguarda *Pregel*. Sono stati selezionati 5 algoritmi su grafi su cui sono stati effettuati i test sulle implementazioni sui 2 framework. Sono stati scelti 4 algoritmi su grafi non orientati e 1 algoritmo su grafo orientato. Gli algoritmi per grafi non orientati sono:

1. **DegreeCalculator**, Calcolo del grado dei vertici.
2. **SSSP**, Cammini minimi da sorgente singola.
3. **NodeIterator++** [?], Algoritmo per il calcolo dei triangoli presenti nel grafo.
4. **Densest Subgraph** [?], Algoritmo per la ricerca del sottografo più denso del grafo, nella sua versione per grafi non orientati.

Per i grafi orientati invece è stata sviluppata la versione dell'algoritmo Densest Subgraph [?] nella sua versione per gradi diretti.

In modello *Pregel*, come visto nel capitolo precedente, è sviluppato su grafi orientati, dove ogni arco è associato alla struttura dati del vertice sorgente. Ogni arco di un grafo non orientato, viene rappresentato da una coppia di archi, uno per ogni direzione. Quindi un arco non orientato  $(u,v)$  è rappresentato dalla coppia di archi orientati  $(u,v)$  e  $(v,u)$ , la prima coppia associata al vertice  $v$  mentre la seconda al vertice  $u$ . Per questo motivo, gli algoritmi su grafi non orientati sono stati sviluppati partendo da un input composto da entrambe le coppie orientate che formano il grafo non orientato.

Per i test è stata necessaria l'implementazione di quasi tutti gli algoritmi scelti. Non è stata necessaria l'implementazione dell'algoritmo *SSSP* per *Pregel*. E' stato possibile sfruttare l'implementazione *SSSP* per grafi pesati presente nel package *giraph-examples* all'interno della distribuzione del framework *Apache Giraph* [?]. Inoltre è stata utilizzata l'implementazione di *NodeIterator++* in *MapReduce* per *Apache Hadoop* [?] implementata da Irene Finocchi, Marco Finocchi, Emanuele G Fusco [?].

In questo capitolo, per ogni algoritmo che si è deciso di utilizzare come base per i test, verrà fornita una descrizione generale dei passi dell'algoritmo e una descrizione

dell'implementazione realizzata per lo sviluppo secondo i modelli *MapReduce* e *Pregel*.

## 4.1 DegreeCalculator

L'algoritmo per il calcolo del grado dei nodi, prende in input un Grafo non orientato  $G = (V, E)$  dove  $V$  sono i vertici del grafo  $G$  ed  $E$  sono gli archi del grafo  $G$ . Per ogni nodo  $v \in V$  l'algoritmo calcola il grado del vertice  $v$ ,  $\deg(v) = |\{(v, u) | (v, u) \in E\}|$ .

### 4.1.1 MapReduce

L'algoritmo *DegreeCalculator* è stato implementato in una singola iterazione *MapReduce*.

La funzione *Map* prende in input la coppia di valori  $\langle v, u \rangle$  che rappresenta un arco del grafo e restituisce in output la coppia  $\langle v, u \rangle$ .

La funzione *Reduce* riceve in input la coppia di valori  $\langle v, list(u) \rangle$  dove  $list(u)$  è la lista di vertici collegati da un arco a  $v$ . La funzione *Reduce* restituisce in output la coppia di valori  $\langle v, degree(v) \rangle$  dove  $degree(v)$  è il grado del nodo  $v$  dato dal numero di vertici presenti in  $list(u)$ .

**Pseudocode 4.1.** Pseudocodice funzione *Map*

```
map(String v, String u){
    // key: vertice di partenza arco
    // value: vertice destinazione
    emit(v, u);
}
```

**Pseudocode 4.2.** Pseudocodice funzione *Reduce*

```
reduce(String v, Iterator vicini){
    // v: ID Vertice v
    // vicini: Lista vertici vicini
    int degree= 0;
    foreach u in vicini{
        degree += 1;
    }
    emit(v,degree);
}
```

### 4.1.2 Pregel

L'implementazione *Pregel* di *DegreeCalculator* implementata è composta da un solo *Superstep*. Il valore associato ad ogni vertice rappresenta il grado del vertice.

Nel *Supertep* la funzione *Compute* conta gli archi associati alla struttura dati del vertice e lo associa al valore associato al vertice. Finito il primo *Superstep* tutti i vertici passano allo stato *Vote to Halt*.

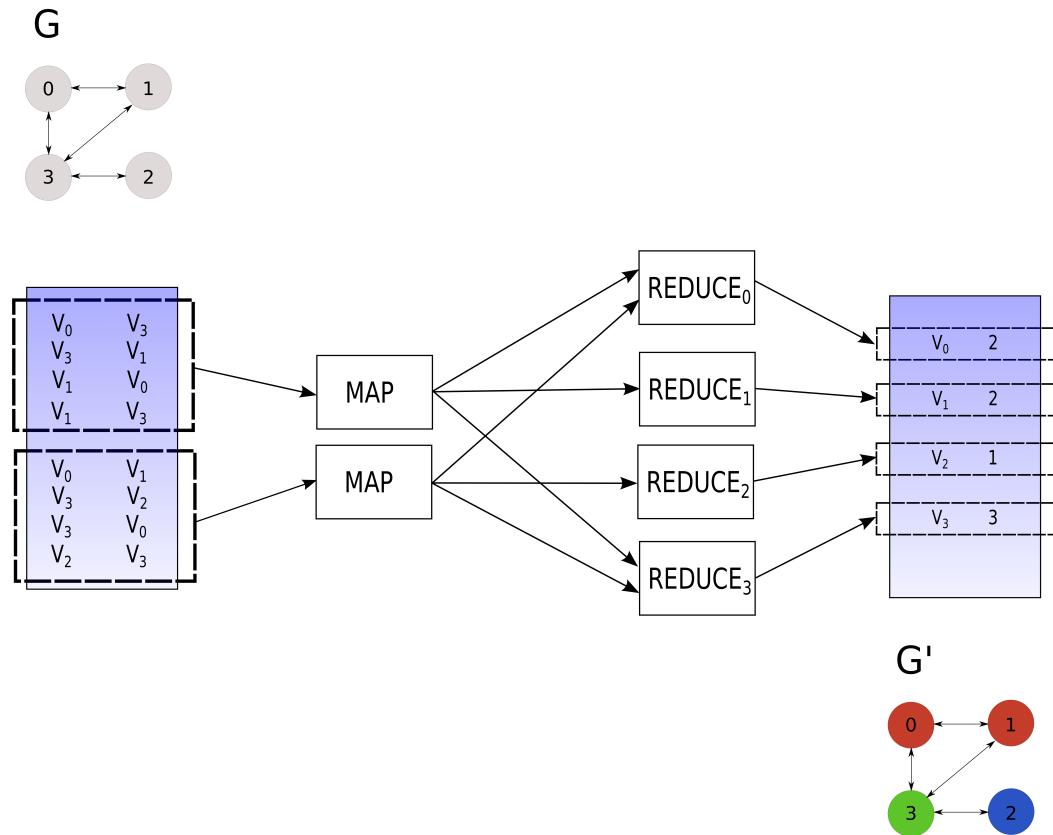


Figura 4.1. Esempio iterazione MapReduce DegreeCalculator

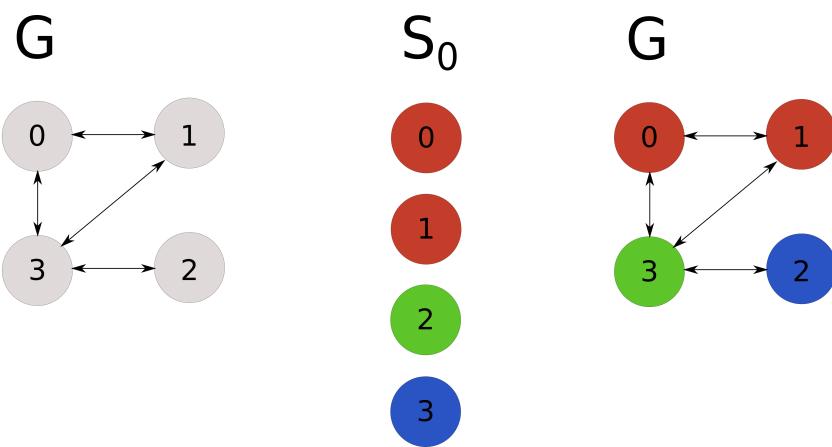


Figura 4.2. Esempio Superstep per DegreeCalculator in Pregel

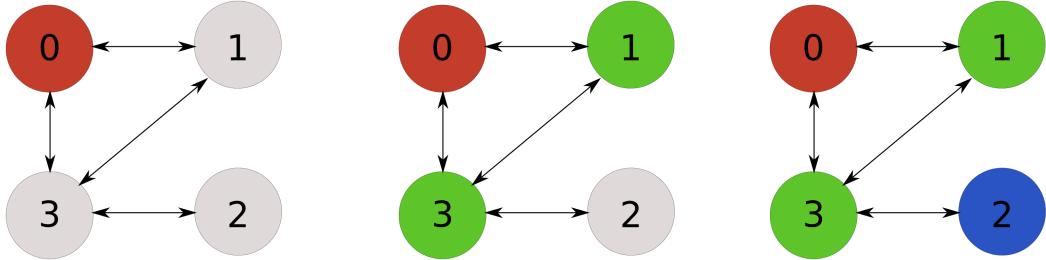


Figura 4.3. Esempio iterazioni SSSP

**Pseudocode 4.3.** Pseudocodice funzione *Compute* in *DegreeCalculator*

```

compute(Vertex vertex, Messages messages){
    degree = 0;
    for each u in vertex.getEdegs(){
        degree += 1;
    }
    vertex.value = degree;
    vertex.voteTohalt();
}

```

## 4.2 SSSP, Cammini minimi da singola sorgente

*Single Source Shortest Path (SSSP)* è l'implementazione per grafi non pesati dell'algoritmo per il calcolo di cammini minimi di Bellman-Ford. Ad ogni vertice del grafo  $v$  è associato un valore  $distanza_v$  inizializzato a  $\infty$  tranne per il nodo sorgente il cui valore è inizializzato a 0.

In figura ?? è rappresentato un esempio di iterazione del algoritmo SSSP su di un grafo. Ad ogni iterazione di *SSSP* ogni nodo  $v$  confronta  $distanza_v$  con i valori  $distanza_u$  dei nodi  $u$  vicini del nodo  $v$ . Il valore del nodo  $v$  viene aggiornato nel corso dell'iterazione se  $distanza_v < \min(distanza_u) + 1$ , con  $\min(distanza_u)$  valore più piccolo tra i valori  $distanza_u$ .

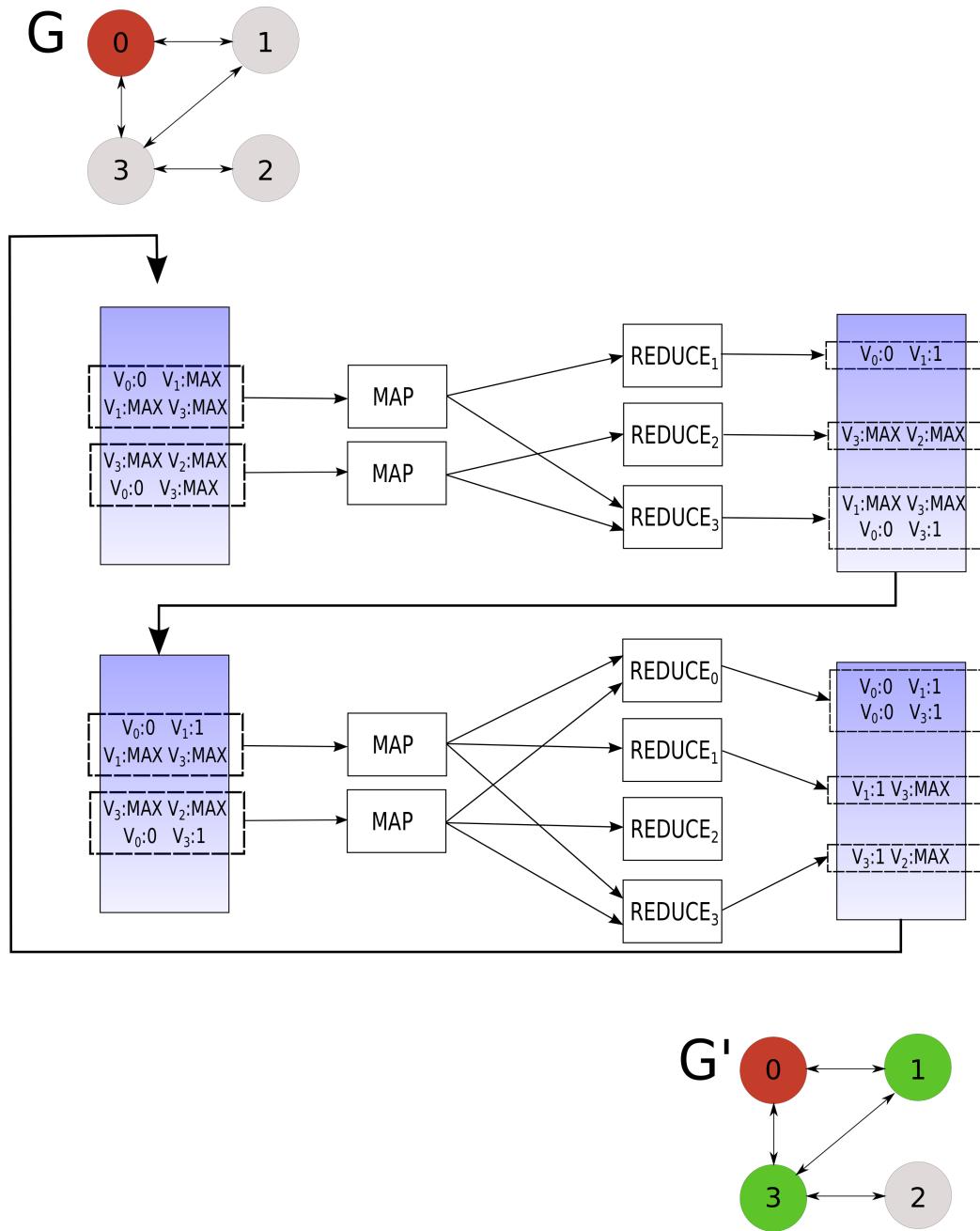
L'algoritmo termina dopo  $|V|$  iterazioni di *SSSP* oppure, se nel corso di un iterazione, nessun vertice del grafo aggiorna il valore *distanza* associato.

### 4.2.1 MapReduce

Per implementare SSSP nel modello *MapReduce*, ogni vertice  $v$  del grafo viene rappresentato nel formato  $ID_v:distanza_v$ . I valori dei vertici  $u$  del grafo sono inizializzati a  $ID_u:\infty$  tranne il nodo  $s$  sorgente inizializzato a  $ID_s:0$ .

Ogni iterazione di *SSSP* è implementata attraverso 2 iterazioni *MapReduce*. L'algoritmo termina quando non vengono più modificati i valori *distanza* dei nodi, si esce dall'iterazione SSSP e si passa ad un ultima iterazione *MapReduce* che produce in output il risultato finale dell'algoritmo. Il risultato prodotto è nel formato  $<ID\_nodo, distanza_{minS}>$ , dove  $distanza_{minS}$  è la distanza minima dal nodo sorgente.

In figura ?? è rappresentato un esempio di un iterazione di *SSSP* in *MapReduce*. La prima iterazione *MapReduce* si occupa di propagare il valore *distanza* sugli archi del



**Figura 4.4.** Esempio delle 2 iterazioni *MapReduce* che implementano un interazione *SSSP*

grafo. La funzione *Map* prende in input la coppia di valori  $\langle v:distanza_v, u:distanza_u \rangle$  e produce in output la coppia  $\langle v:distanza_v, u:distanza_u \rangle$ . La funzione *Reduce* riceve in input la coppia di valori  $\langle v:distanza_v, list(u:distanza_u) \rangle$ , dove  $list(u:distanza_u)$  è la lista di nodi  $u$  raggiungibili dal nodo  $v$ . *Reduce* produce in output, per ogni elemento  $w \in list(u:distanza_u)$  la coppia di valori  $\langle v:distanza_v, u:distanza_{min} \rangle$ , dove il valore  $distanza_{min}$  è calcolato come il valore minore tra  $distanza_i$  ed il minimo tra valori  $distanza_u + 1$  presenti in  $list(u:distanza_u)$ .

**Pseudocodice 4.4.** Pseudocodice funzione *Map* prima iterazione *MapReduce* di *SSSP*

```

map(String v, String u){
    // v: vertice sorgente
    // u: vertice destinazione
    Emit(v, u);
}

```

**Pseudocodice 4.5.** Pseudocodice funzione *Reduce* prima iterazione *MapReduce* di *SSSP*

```

reduce(String v, Iterator vicini){
    // v: ID Vertice
    // vicini: Lista vertici vicini
    int dist_min = v.distanza;
    for each w in vicini{
        if(dist_min > w.distanza + 1){
            dist_min = w.distanza + 1;
        }
    }
    for each w in values{
        emit(v, w:dist_min+1)
    }
}

```

La seconda iterazione *MapReduce* è necessaria per permettere di aggiornare i valori *distanza* relativi ai nodi sorgenti degli archi, dopo la prima iterazione sono aggiornati soltanto i nodi destinazione.

La funzione *Map* della seconda iterazione prende in input gli archi nel formato  $\langle v:distanza_v, u:distanza_u \rangle$  e produce il doppio output formato dalle coppie  $\langle v, u:distanza_u:distanza_v:S \rangle$  e  $\langle u, v:distanza_v:distanza_u:D \rangle$ ,  $S$  e  $D$  sono dei valori segnaposto,  $S$  segnala che il nodo  $v$  è un nodo sorgente e  $D$  che  $v$  è invece nodo destinazione. I valori sentinella vengono utilizzati nella funzione *Reduce* per ricostruire il corretto verso degli archi del grafo.

La funzione *Reduce* della seconda fase prende in input la coppia di valori  $\langle v, list(u:distanza_u:distanza_v:\$) \rangle$  con  $\$ \in [S, D]$ . Per ogni valore presente nella lista  $\langle v, list(u:distanza_u:distanza_v:\$) \rangle$  con  $\$ = S$ , la funzione *Reduce* produce in output la coppia di valori  $\langle v:distanza_{minima}: u:distanza_u \rangle$ , dove  $distanza_{minima}_i$  è il minore dei valori  $distanza_i$  in  $list(v_j:distanza_j:distanza_i:\$)$  con  $\$ = D$ .

Le iterazioni *SSSP* terminano quando non vengono aggiornati i valori *distanza* nella corso della prima *MapReduce*.

**Pseudocode 4.6.** Pseudocodice funzione *Map* SSSP

```

|| map(String v, String u){
||   // v: vertice sorgente
||   // u: vertice destinazione
||   Emit(v.ID, u : v.distanza : S);
||   Emit(u.ID, v : u.distanza : D);
|| }
```

**Pseudocode 4.7.** Pseudocodice funzione *Reduce* SSSP

```

|| reduce(String v, Iterator vicini){
||   // v: ID Vertice
||   // vicini: Lista vertici vicini
||   int dist_min = v.distanza;
||   for each w in vicini{
||     if(w.$ == S){
||       if(dist_min > w.distanza){
||         dist_min = w.distanza;
||       }
||     }
||   }
||   for each w in vicini{
||     if(w.$ == D){
||       emit(v:dist_min,w)
||     }
||   }
|| }
```

L'ultima iterazione *MapReduce*, che salva i risultati in output, viene eseguita quando sono terminate le iterazioni *SSSP*. La funzione *Map* prende in input la coppia di valori  $\langle v:distanza_v, u:distanza_u \rangle$  e produce in output la coppia  $\langle v:distanza_v, \varepsilon \rangle$ , dove  $\varepsilon$  è stringa vuota.

La funzione *Reduce* prende in input la coppia di valori  $\langle v:distanza_v, list(\varepsilon) \rangle$  e produce in output la coppia  $\langle v, distanza_v \rangle$  con  $distanza_v$  valore del cammino minimo dal nodo sorgente al nodo  $v$ .

**Pseudocode 4.8.** Pseudocodice funzione *Map* SSSP

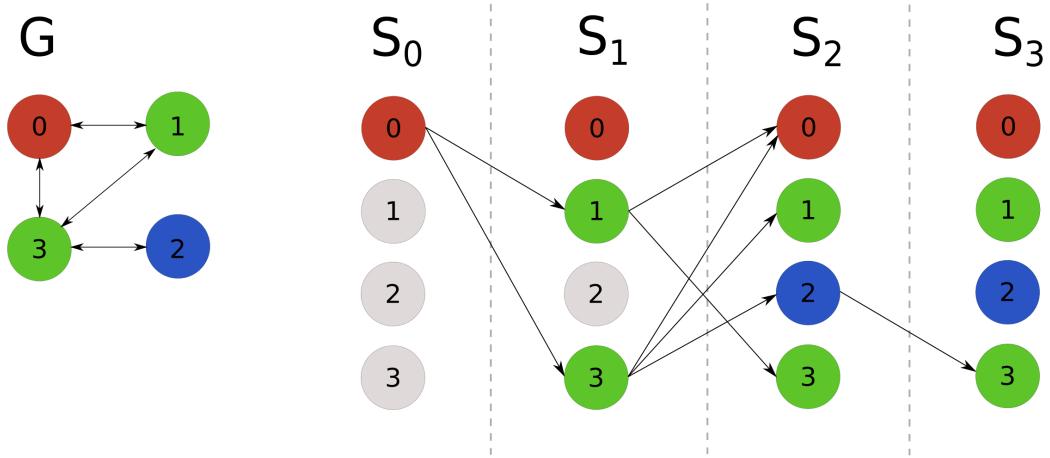
```

|| map(String Vs, String Vd){
||   // Vs: vertice di partenza
||   // Vd: vertice destinazione
||   Emit(Vs, "");
|| }
```

**Pseudocode 4.9.** Pseudocodice funzione *Reduce* SSSP

```

|| reduce(String Vs, Iterator Vd){
||   // Vs: ID Vertice
||   // Vd: Lista vertici vicini
||   emit(Vs.ID, Vs.distanza)
|| }
```



**Figura 4.5.** Esempio SSSP in Pregel

#### 4.2.2 Pregel

L'implementazione in *Pregel* sviluppata per realizzare l'algoritmo *SSSP* è costituita da un primo *Superstep* dove il valore associato al vertice  $v$ , che rappresenta il valore  $distanza_v$  associato a  $v$ , viene inizializzato a  $\infty$  tranne per il vertice sorgente il cui valore è inizializzato a 0.

Nella figura ?? è rappresentato un esempio di computazione in *Pregel* di *SSSP*.

Ad ogni *Superstep*  $S$  è eseguita un'iterazione *SSSP* in cui un vertice  $v$  riceve i messaggi, inviati nel *Superstep*  $S-1$ , messaggi che contengono il valore  $distanza_u$  dei vertici  $u$  vicini di  $v$ . Il vertice  $v$  confronta i messaggi ricevuti con il proprio valore  $distanza_v$ . Nel caso che almeno uno dei messaggi contiene un valore  $distanza_u$  che risulta minore di  $distanza_v$ , il valore di  $distanza_v$  viene aggiornato con il valore minore contenuto nei messaggi. Nel caso il vertice  $v$  viene aggiornato nel *Superstep*  $S$ , viene inviato un messaggio contenente il nuovo valore  $distanza_v + 1$  ai vertici vicini di  $v$ , che verrà recapitato nel *Superstep* successivo  $S+1$ . Tutti i vertici passano allo stato *Vote to Halt* e solo i vertici a cui è stato inviato un messaggio in  $S+1$  saranno nello stato *Active* in  $S+1$ . La computazione di *SSSP* termina quando nessun vertice invia messaggi nel corso del *Superstep* e quindi nessuno dei vertici del grafo ha aggiornato il valore *distanza*.

**Pseudocode 4.10.** Pseudocodice funzione *Compute SSSP*

```

compute(Vertex vertex, Messages messages){
    if (Superstep == 0) {
        vertex.value = MAX_VALUE;
        if(vertex == sorgente){
            vertex.value = 0;
        }
    }
    distanza_min = vertex.value;
    for each m in messages{
        minDist = min(minDist, m);
    }

    if (minDist < vertex.value) {
        //Valore distanza aggiornato
        vertex.value = minDist;
        for each edge in vertex.getEdegs(){
            distance = minDist + 1;
            sendMessage(edge.ID, minDist + 1);
        }
    }
    vertex.voteToHalt();
}
}

```

### 4.3 NodeIterator++, Calcolo del numero di triangoli del grafo

NodeIterator++ [?] è un algoritmo per il calcolo del numero di triangoli in un grafo in modo efficiente nel modello MapReduce [?]. L'idea di base dell'algoritmo è di costruire per ogni nodo  $v$  del grafo, una lista di tutte le possibili coppie  $(u,w)$  di nodi vicini a  $v$ . Per ogni coppia di nodi  $(u,w)$ , se  $(u,w) \in E$ , con  $E$  insieme degli archi del grafo, allora esiste il triangolo  $(v,u,w)$ .

Nella versione base dell'algoritmo, *NodeIterator*, vengono prese in esame tutte le coppie possibili di nodi vicine di un nodo  $v$ , questo portava a contare, in modo molto inefficiente, ogni triangolo 6 volte. Per migliorare l'efficienza e contare ogni triangolo 1 volta, la versione *NodeIterator++* dell'algoritmo, introduce un relazione di ordine totale sui vertici del grafo, *prec* ( $\prec$ ). Per una coppia di nodi del grafo  $u$  e  $w$  del grafo, si ha che  $u \prec w$  se  $\deg(u) < \deg(w)$ , dove  $\deg(v)$  è il grado del nodo  $v$ . In caso di nodi con lo stesso grado si sceglie in modo arbitrario e consistente l'ordine di precedenza tra i nodi, come per esempio l'ordine sul valore numerico corrispondente all'ID del vertice.

**Pseudocode 4.11.** Pseudocode NodeIterator++

```

T = 0;
for each v in V{
    for each u in vicinato(v) && u prec v{
        for each w in in vicinato(v) && w prec u{
            if((u, w) is Edge){
                T = T + 1;
            }
        }
    }
}

```

### 4.3.1 MapReduce

Per testare l'algoritmo *NodeIterator++* in *MapReduce* ho utilizzato l'implementazione realizzata da Irene Finocchi, Marco Finocchi, Emanuele G Fusco per [?].

Per sviluppare *NodeIterator++* sono necessari due iterazione *MapReduce*. Nella prima iterazione vengono generate tutte le coppie di vicini di un nodo  $v$ ,  $(u,v)$ .

La funzione *Map* riceve in input un arco del grado nella forma  $\langle v, u \rangle$  e produce in output la coppia di vertici  $\langle v, u \rangle$  soltanto se  $v \prec u$ .

La funzione *Reduce* riceve in input la coppia  $\langle v, list(u) \rangle$ , con  $list(u)$  lista di nodi vicini di  $v$  per cui  $v \prec u$ . Per ogni coppia di elementi di  $list(u)$ ,  $(u,w)$  produce in output la coppia di valori  $\langle v, (u,w) \rangle$ .

**Pseudocode 4.12.** Pseudocode Map Fase 1 NodeIterator++

```

map(String v, String u){
    // key: vertice di partenza
    // value: vertice destinazione
    if(v prec u){
        Emit(v, u);
    }
}

```

**Pseudocode 4.13.** Pseudocode Reduce Fase 1 NodeIterator++

```

reduce(String key, Iterator values){
    // key: ID Vertice
    // values: Lista vertici vicini
    for each u in values{
        for each w in values{
            if(u prec w){
                emit(v, (u,w))
            }
        }
    }
}

```

Nella seconda fase *MapReduce* vengono verificate quali tra le coppie di nodi, generate dalla prima iterazione *MapReduce*, hanno un arco che collega tra loro i 2 vertici e che quindi costituiscono un triangolo del grafo.

La funzione *Map* prende in input i valori da due differenti sorgenti: l'output del round precedente,  $\langle v, (u, w) \rangle$  e la lista di archi  $\langle v, u \rangle$  dall'input iniziale del iterazione *SSSP*. Per il primo tipo di input è prodotto in output la coppia di valori  $\langle (u, w), v \rangle$ , per il secondo tipo di input, invece, è prodotto in output la coppia di valori  $\langle (v, u), \$ \rangle$ , dove  $\$$  è un simbolo segnaposto. Il simbolo  $\$$  è utilizzato poi nella funzione di *Reduce* come simbolo che segnala che la coppia di nodi  $(v, u)$  è collegata da un arco del grafo.

La funzione *Reduce* prende in input la coppia  $\langle (u, v), list(w) \rangle$ , dove  $(u, v)$  è un arco del grafo e  $list(w)$  è una lista di elementi  $w$ , dove l'elemento  $w$  o è un vertice del grafo o è il valore segnaposto  $\$$ . Se nella lista  $list(w)$  è presente il valore  $\$$  allora l'arco  $(u, v) \in E$ , con  $E$  insieme degli archi del grafo. In questo caso ogni valore elemento  $list(w)$ ,  $w$  diverso da  $\$$ , rappresenta un nodo del grafo che ha come vicino sia il nodo  $u$  che il nodo  $w$ , per cui la presenza dell'arco  $(u, v)$  chiude il triangolo  $(u, v, w)$ . L'output della funzione *Reduce*, nel caso  $\$ \in list(w)$ , è la coppia di valori  $\langle (u, v), T \rangle$ , dove  $T$  è il numero di triangoli chiusi dalla coppia  $(u, v)$ . Nel caso in cui  $\$ \notin list(w)$  allora la coppia  $(u, v)$  non ha un arco che collega i 2 vertici, nessuno dei possibili triangolo segnalati in  $list(w)$  è chiuso e la funzione non produce nessun valore in output.

Per contare tutti i triangoli rilevati è necessario un'ulteriore passaggio, implementabile sempre in *MapReduce*, in cui vengono contate le somme parziali dei triangoli rilevate delle funzioni *Reduce* nell'iterazione precedente. La funzione *Map* mappa tutte le coppie di valori del tipo  $\langle (u, v), T \rangle$  verso un'unica funzione *Reduce* utilizzando un'unica chiave. La funzione *Reduce* riceve tutti i risultati parziali  $T$  ricavati nel passo precedente, li somma tra loro e fornisce in output il risultato totale dei triangoli presenti nel grafo.

In figura ?? è rappresentato un esempio della computazione di *NodeIterator++* in *MapReduce*.

**Pseudocode 4.14.** Pseudocodice Map Fase 2 NodeIterator++

```

map(String v, String u){
    // key: vertice di partenza
    // value: vertice destinazione
    if(input <v, (u,w)>){
        Emit((u,w), v);
    }
    if(input <v, u>){
        Emit((v,u), $);
    }
}

```

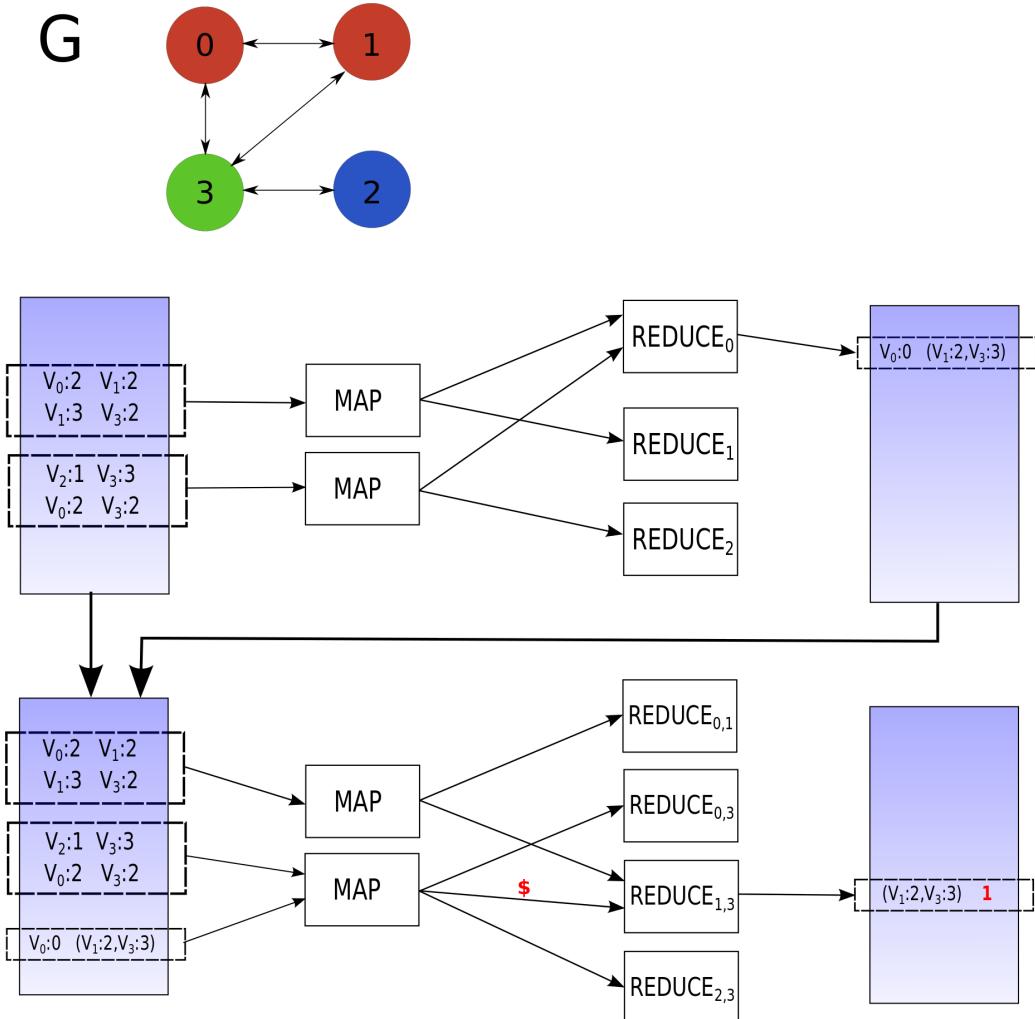


Figura 4.6. Esempio calcolo *NodeIterator++* in *MapReduce*

Pseudocode 4.15. Pseudocode Reduce Fase 2 NodeIterator++

```

reduce(String key, Iterator values){
    // key: ID Vertice
    // values: Lista vertici vicini
    for each u in values{
        for each w in values{
            if(u prec w){
                emit(v, (u,w))
            }
        }
    }
}

```

### 4.3.2 Pregel

L'implementazione realizzata per *NodeIterator++* su *Pregel* prevede, per ogni vertice  $v$  del grafo, l'esecuzione di 4 *Superstep* :

1. Nel primo *Supertep* il vertice  $v$  calcola ed invia a tutti i nodi vicini il valore  $\deg(v)$ , grado di  $v$ .
2. Nel secondo *Superstep* il vertice  $v$  confronta il proprio grado con il grado dei nodi ricevuti utilizzando la funzione di ordine totale  $\prec$  definita in *NodeIterator++*. Per ogni coppi di archi orientati, che compongono l'arco non orientato in *Pregel*, viene eliminato l'arco con il verso che non rispetta l'ordine  $\prec$ . Come nell'esempio in ??, nel grafo  $G'$ , dopo il secondo *Superstep*  $S_1$ , rimangono solo gli archi che rispettano l'ordine definito da  $\prec$ , come ad esempio  $(0,1)$  e  $(1,3)$ .
3. Nel terzo *Superstep*, il vertice  $v$  invia a tutti i nodi, collegati a  $v$  da un arco del grafo ottenuto dopo l'iterazione precedente, la lista dei nodi vicini.
4. Nel ultimo *Supertep*, il vertice  $v$  riceve tramite messaggi la lista dei nodi dai vertici  $w$  a cui è vicino,  $list(w)$ . Nel caso in cui l'arco  $(v,u)$  è un arco del grafo, con  $u \in list(w)$ , allora nel grafo è presente il triangolo  $(u, v, w)$ . Il vertice  $v$  confronta, quindi, la lista dei nodi ricevuta con l'insieme di vertici raggiungibili da  $v$ . Per ogni corrispondenza segnala il triangolo individuato tramite l'utilizzo di un aggregatore di tipo *SUM*, visto nel capitolo introduttivo su *Pregel*.

Alla fine del *Superstep*, secondo il funzionamento degli aggregatori, il nodo Master raccoglie tutti i valori aggregati ottenendo il numero totale di triangoli presenti nel grafo.

**Pseudocode 4.16.** Pseudocode NodeIterator++ Pregel

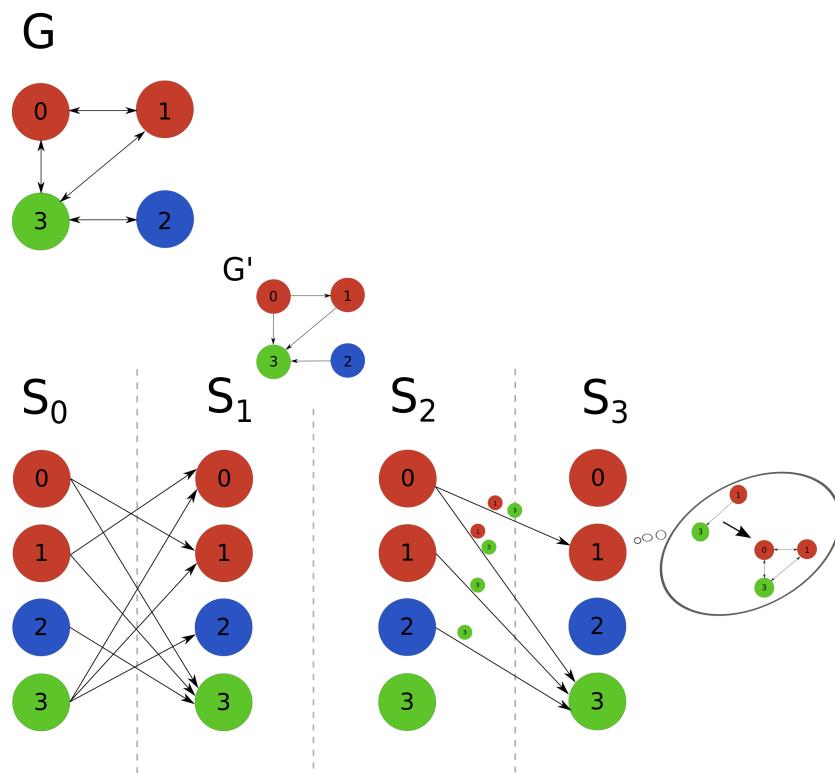
```

compute(Vertex vertex, Messages messages) {
    if (getSuperstep() == 0) {
        degree = vertex.getNumEdges();

        for each edge in vertex.getEdegs(){
            sendMessage(edge.ID, degree);
        }
    } else if (getSuperstep() == 1) {

        for each m in messages{
            if (vertex.ID prec m.ID) {
                this.removeEdge(m.ID, vertexID);
            }
        }
    } else if (getSuperstep() == 2) {
        for each1 edge in vertex.getEdegs(){
            for each2 edge in vertex.getEdegs(){
                sendMessage(edge1.ID, edge2.ID);
            }
        }
    } else if (getSuperstep() == 3) {
        for each m in messages{
            if (m is Edge) {

```



**Figura 4.7.** Esempio calcolo NodeIterator++ in Pregel

```
    T++ ;  
    }  
}  
aggregate(T);  
vertex.voteToHalt();  
}  
}
```

#### 4.4 Densest Subgraph , Ricerca del sottografo più denso in grafi non orientati

L'algoritmo per la ricerca del sottografo più denso [?] trova il sottografo  $G'$  del grado  $G$  la cui densità  $\rho(G')$  è una  $(2 + 2\epsilon)$ -approssimazione di  $\rho^*(G)$ , dove  $\rho^*(G)$  è la densità maggiore tra tutti i sottografi di  $G$ .

L'algoritmo per la ricerca del sottografo più denso per grafi non orientati prende in input un grafo  $G = (V, E)$  non orientato e un valore  $\epsilon > 0$ . L'algoritmo procede in modo iterativo sul grafo. Ad ogni passo l'algoritmo prevede:

- Il calcolo della densità corrente  $\rho$  del grafo  $G'$  è di un valore *soglia*. La densità del grafo  $\rho(G')$  è definita come  $\rho(G') = |E(G')|/|V(G')|$  mentre il valore *soglia* è definito da  $soglia = 2(1 + \epsilon)\rho(G')$ .

2. Ogni nodo  $v$  con  $\deg(v) < \text{soglia}$ , dove  $\deg(v)$  è il grado del vertice  $v$  nel grafo corrente  $G'$ , viene rimosso assieme agli archi ad esso associati ottenendo il grafo  $G$ .
3. Se  $|V(G)| > 0$ , con  $|V(G)|$  numero di archi in  $G$  allora si ripete l'iterazione dal passo 1, altrimenti termina la computazione e viene restituito in output il sottografo con valore densità  $\rho$  maggiore.

In [?] viene dimostrato che il sottografo più denso tra i sottografi  $G$  ottenuti dall'algoritmo è un sottografo la cui densità è  $\rho(G'')$  è una  $(2 + 2\epsilon)$ -approssimazione di  $\rho^*(G)$ , dove  $\rho^*(G)$  è la densità maggiore tra tutti i sottografi di  $G$ .

**Pseudocode 4.17.** Pseudocodice Denesest Subgraph per grafi non orientati

```

G = (V, E)
epsilon > 0
S = V
while (S not EMPTY){
    for each v in S{
        if ( deg(v) <= 2 * (1 + epsilon) * rho(S) ){
            S = S - {v}
        }
    }
    if ( rho(S) > rho*(S) ){
        S* = S
    }
}
return S*;
    
```

#### 4.4.1 MapReduce

Per realizzare l'algoritmo in *MapReduce* è stato implementata l'idea descritta in [?]. Ogni iterazione dell'algoritmo viene realizzata in 3 fasi *MapReduce*.

In figura ?? è rappresentato un esempio dell'esecuzione di un iterazione dell'algoritmo in MapReduce.

Nella prima fase viene calcolato il grado di ogni nodo  $v$  del grafo in input. La realizzazione di questa fase è la stessa dell'implementazione di *DegreeCalculator* in *MapReduce*. Contestualmente alla funzione *Map* della prima fase vengono contati anche il numero totale di vertici e di archi del grafo con cui è calcolata la densità corrente del grafo e del valore *soglia*.

Tra la prima fase e la successiva è effettuato il controllo per cui, se il valore densità è il risultato migliore ottenuto, viene salvato lo stato del sottografo corrente.

Nella seconda e terza fase *MapReduce* vengono eliminati gli archi  $(u, v)$  dove almeno uno tra i vertici  $u$  e  $v$  è di grado minore del valore *soglia*. Nella seconda fase vengono controllati i nodi sorgenti di ogni arco mentre nella fase successiva sono i nodi destinazione degli archi ad essere controllati.

La funzione *Map* della seconda fase prende in input sia l'input iniziale dell'iterazione dell'algoritmo che l'output generato dalla prima fase *MapReduce*. *Map* produce in output la coppia  $\langle v, u \rangle$  per il primo tipo input. Per il secondo tipo di input, nel formato  $\langle v, \deg(v) \rangle$ , viene prodotto in output i valori  $\langle v, \$ \rangle$  solo nel caso che

$\deg(v) > soglia$ . in questo caso il simbolo  $\$$  viene interpretato dalla funzione *Reduce* e significa che l'arco non deve essere eliminato in questa iterazione dell'algoritmo.

La funzione *Reduce* riceve in input la coppia valori  $\langle v, list(u) \rangle$ , dove  $list(u)$  è la lista di nodi raggiungibili da  $v$ . Per ogni elemento  $w \in list(u)$ , *Reduce* produce in output la coppia  $\langle v, w \rangle$  solo nel caso in cui  $\$ \in list(w)$ .

La terza fase *MapReduce* opera in modo simile alla seconda, al posto di prendere in input il grafo iniziale dell'iterazione, prende in input l'output della fase precedente. Per controllare il grado dei nodi destinazione degli archi del grafo, la funzione *Map*, invece di produrre i valori  $\langle v, u \rangle$ , produce in output la coppia  $\langle u, v \rangle$ . L'output di *Map* per il secondo tipo di input è lo stesso prodotto nella fase *MapReduce* precedente.

Anche la funzione *Reduce* opera in modo simile alla funzione *Reduce* della fase precedente con la differenza che in output, per ogni elemento  $w \in list(u)$ , produce in output la coppia di valori  $(w, v)$ , sempre soltanto nel caso che sia presente il simbolo  $\$$  in  $list(u)$ .

**Pseudocode 4.18.** Pseudocodice funzione *Map* fase 2 *MapReduce* di iterazione Denesest  
Subgraph per grafi non orientati

```

map(String v, String u){
    // v: vertice di partenza
    // u: vertice destinazione
    if(input <v, deg(v) >){
        if( deg(v) > soglia){
            Emit(v, $);
        }
    }
    if(input <v, u>){
        Emit(v, u);
    }
}

```

**Pseudocode 4.19.** Pseudocodice funzione *Reduce* fase 2 *MapReduce* di iterazione Denesest  
Subgraph per grafi non orientati

```

reduce(String v, Iterator vicini){
    // v: ID Vertice
    // vicini: Lista vertici vicini
    if($ in vicini){
        for each u in vicini{
            emit(v,u);
        }
    }
}

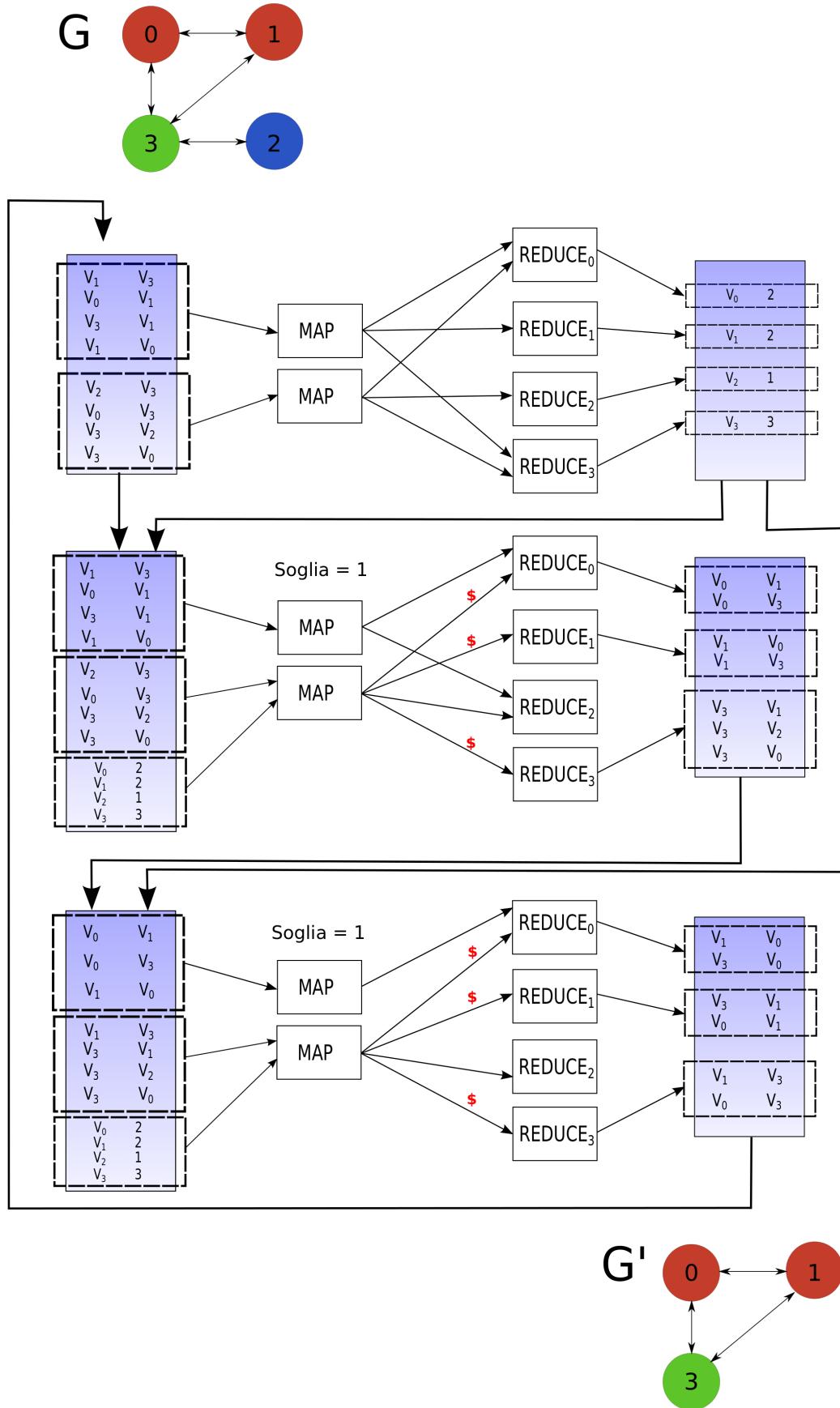
```

#### 4.4.2 Pregel

Nell'implementazione *Pregel* di [?], ogni iterazione dell'algoritmo è composta da 2 *Superstep*.

Nella figura ?? è rappresentato un esempio di iterazione dell'algoritmo in *Pregel*.

Nel primo *Superstep* il vertice  $v$ , utilizzando le informazioni statistiche del grafo mantenute costantemente dal modello *Pregel*, calcola i valori correnti di densità



**Figura 4.8.** Esempio calcolo Densest Subgraph per grafi non orientati in MapReduce con soglia indicativa

del grafo,  $\rho(G')$ , e *soglia*. Se il grado del  $v$  non rispetta il valore di *soglia*, cioè se  $\deg(v) < \text{soglia}$ , il vertice  $v$  e tutti gli archi  $(v,u)$ , in cui il nodo  $v$  è sorgente, vengono eliminati dal grafo. Visto che in *Pregel*, come visto in precedenza, gli archi non orientati sono composti da 2 archi orientati ed la struttura di ogni arco è mantenuta dal vertice sorgente dall'arco, tramite messaggio viene notificata l'eliminazione dell'arco al nodo destinazione.

Nel secondo *Superstep* i nodi che ricevono i messaggi, che non sono stati già eliminati nel passo precedente, eliminano il verso opposto degli archi segnalati ed eliminati nel primo *Superstep*.

Alla fine di ogni iterazione viene controllata la densità corrente dal nodo *Master* dell'architettura *Pregel*, ogni volta che si ottiene un valore *densità* migliore viene salvato il sotto-grafo relativo.

La computazione termina quando non ci sono più vertici attivi nel grafo.

**Pseudocode 4.20.** Pseudocodice Denesest Subgraph per grafi non orientati

```

compute(Vertex vertex, Messages messages){
    if (Superstep % 2 == 0) {

        rho = |E(G)| / |V(G)| ;
        soglia = 2 * (1 + epsilon) * rho;

        if(deg(v) <= soglia){
            elimina vertex;
            for each edge in vertex.getEdegs(){
                elimina edge;
                sendMessage(edge.ID, vertex.ID);
            }
        }
    }

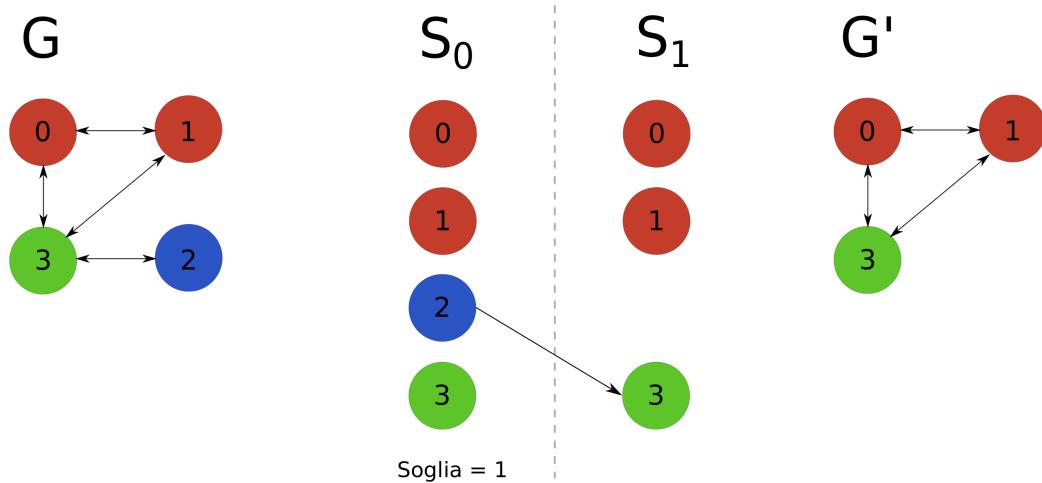
    } else if (Superstep % 2 == 1) {

        for each m in messages{
            for each edge in vertex.getEdegs(){
                if(edge.ID == m)
                    elimina edge;
            }
        }
    }
}

```

## 4.5 Densest Subgraph, Ricerca del sottografo più denso in grafi orientati

La versione dell'algoritmo per la ricerca del sottografo più denso [?] per grafi diretti procede in modo simile alla versione per gradi non diretti. Come per grafi non diretti, l'algoritmo raggiunge, nel caso peggiore, una  $2(1+\epsilon)$ -approssimazione del valore della densità maggiore tra tutti i sottografi del grafo  $G$ ,  $\rho^*(G)$ .

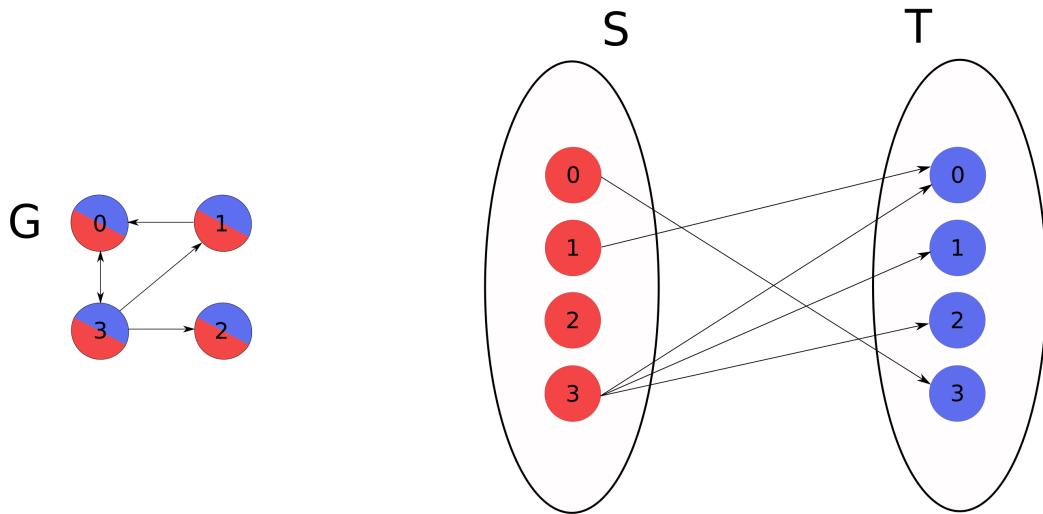


**Figura 4.9.** Esempio iterazione Densest Subgraph per grafi non orientati in Pregel

L'algoritmo per la ricerca del sottografo più denso per grafi orientati prende in input un grafo  $G = (V, E)$  non orientato, un valore  $\epsilon > 0$  ed un valore  $c > 0$ . L'algoritmo procede in modo iterativo, ad ogni passo dell'algoritmo:

1. E' calcolata la densità corrente del grafo  $G'$ , per un grafo diretto la densità è calcolata come  $\rho(G') = |E(S, T)| / \sqrt{|S||T|}$ , con  $S, T \subseteq V$ ,  $E(S, T) = E \cap (S \times T)$ . Gli insieme  $S$  e  $T$  sono inizializzati in  $S = T = V$ . Nella figura ?? è mostrato un esempio di come vengono ripartiti i vertici del grafo  $G$  sugli insieme  $S$  e  $T$ .
2. Se  $|S| / |T| > c$  allora vengono eliminati vertici dall'insieme  $S$  il cui grado uscente è minore del valore *soglia*, altrimenti vengono eliminati vertici dall'insieme  $T$  il cui grado entrante è minore del valore *soglia*. Nel primo caso sia ha  $soglia = (1 + \epsilon)|E(S, T)| / |S|$  mentre nella seconda caso  $soglia = (1 + \epsilon)|E(S, T)| / |T|$ .
3. Se nel grafo  $G$  ottenuto sono ancora presenti vertici in una dei due insieme  $S$  o  $T$ , l'iterazione riprende dal passo 1.

In [?] viene dimostrato che uno dei grafi  $G$  è una  $(2\epsilon + 2)$ -approssimazione del sottografo più denso del grafo iniziale  $G$ .



**Figura 4.10.** Esempio ripartizione vertici di  $G$  sugli insiemi  $S$  e  $T$

**Pseudocode 4.21.** Pseudocodice Denesest Subgraph per grafi orientati

```

G = (V, E)
epsilon > 0
c > 0
S = V
T = V
while (S not EMPTY AND T not EMPTY){
    if( |S|/|T| > c){
        for each v in S{
            soglia_S = (1 + epsilon) * |E(S,T)| / |S|
            if ( deg_u(v) <= soglia_S ) {
                S = S - {v}
            }
        }
    }else{
        for each u in T{
            soglia_T = (1 + epsilon) * |E(S,T)| / |T|
            if ( deg_e(u) <= soglia_T ){
                T = T - {v}
            }
        }
    }

    if ( rho(S,T) > rho*(S,T) ){
        S* = S
        T* = T
    }
}
return S*, T*;

```

### 4.5.1 MapReduce

Ogni iterazione dell'algoritmo per la ricerca del sottografo più denso in grafi orientati è stata sviluppata attraverso 3 iterazioni *MapReduce*. Nelle prime due iterazioni è calcolato il grado uscente dei vertici che si trovano nell'insieme  $S$  e il grado entrante dei vertici che appartengono all'insieme  $T$ . La terza iterazione *MapReduce* varia a seconda che si vadano ad eliminare vertici dall'insieme  $S$  oppure ad eliminare i vertici dell'insieme  $T$ .

Nella figura ?? è rappresentato un esempio di iterazione in *MapReduce* dell'algoritmo.

Entrambe le funzioni *Map* relative alla prima e seconda iterazione *MapReduce* prende in input la coppia di valori  $\langle v, u \rangle$  arco del grafo  $G$ . Nel primo caso, in cui è calcolato il grado uscente dei vertici nell'insieme  $S$ , *Map* restituisce in output la coppia di vertici  $\langle v, u \rangle$ . Nella seconda iterazione, in cui è calcolato il grado uscente dei vertici in  $T$ , l'output della funzione *Map* è  $\langle u, v \rangle$ .

Le funzioni *Reduce* di entrambe le prime 2 fasi *MapReduce* ricevono valori in input nel formato  $\langle v, list(u) \rangle$ , dove  $list(u)$  sono i nodi  $u$  del grafo corrente collegati da un arco (entrante o uscente) al vertice  $v$ . L'output restituito dalla funzione è  $\langle v, deg(v) \rangle$  dove  $deg(v)$  è il numero di vertici in  $list(u)$ , che rappresenta, a seconda della fase *MapReduce* il grado uscente o entrante di  $v$ .

Contestualmente alle prime due fasi vengono contati il numero di vertici presenti nei due insieme,  $|S|$  e  $|T|$ , e il numeri di archi  $|E(S, T)|$ . Inoltre è calcolato la densità  $\rho(G')$  dove  $G'$  è lo stato del grafo nell'iterazione corrente. Nel caso  $\rho(G')$  è il valore densità più alto rilevato fino a quel momento viene salvato lo stato del grafo  $G'$ .

Se  $|S| / |C| > c$  allora è eseguita l'iterazione *MapReduce* che elimina vertici dall'insieme  $S$ , altrimenti è eseguita l'iterazione *MapReduce* in cui sono eliminati vertici dall'insieme  $T$ . Entrambe le iterazione prendono 2 differenti dati in input: l'input iniziale contente la lista di archi del grafo  $G$ , nella forma  $\langle v, u \rangle$ , e l'output di una delle 2 iterazioni in cui è calcolato il grado dei nodi degli insiemi  $S$  o  $T$ , a seconda di qual'è l'insieme in cui si andrà a controllare il grado dei vertici. La funzione *Map* per il primo input (nel formato  $\langle v, u \rangle$ ) restituisce in output la coppia di valori  $\langle v, u \rangle$  nel caso si controllano i vertici in  $S$  ( $\langle u, v \rangle$  nel caso del controllo dei vertici in  $T$ ). Per il secondo input (nel formato  $\langle v, deg(v) \rangle$ ) se  $deg(v) < soglia$  viene generato in output la coppia di valori  $\langle v, \$ \rangle$ , dove  $\$$  è un valore segnaposto.  $\$$  ha la funzione di segnalare alla funzione *Reduce* che il vertice  $v$  ha un grado maggiore del valore *soglia* corrente. Il valore *soglia* dipende dall'insieme,  $S$  o  $T$ , che è in esame nell'iterazione corrente dell'algoritmo.

La funzione *Reduce* prende in input la coppia di valori  $\langle v, list(u) \rangle$  dove  $list(u)$  sono i vertici collegati da un arco entrante a  $v$  nel caso in cui è in esame l'insieme  $S$  (o da un arco uscente nel caso dell'insieme  $T$ ). In output la funzione *Reduce* restituisce in output, per ogni elemento  $w \in list(u)$  i valori  $\langle v, w \rangle$  o  $\langle v, w \rangle$  nel caso siano in esame, rispettivamente, gli archi uscenti da un nodo in  $S$  o gli archi entranti nei nodi in  $T$ . In entrambi i casi la funzione *Reduce* restituisce valori solo se il vertice  $v$  rispetta la soglia imposta dall'algoritmo, cioè se il valore segnaposto  $\$ \in list(u)$ .

L'iterazione termina quando non ci sono più nodi negli insieme  $S$  e  $T$ .

**Pseudocodice 4.22.** Pseudocodice funzione *Map* della fase 3 *MapReduce* dell'iterazione Denesest Subgraph per grafi non orientati

```

map(String v, String u){
    // key: vertice di partenza
    // value: vertice destinazione
    if(input <v, deg(v) >{
        if( deg(v) > soglia){
            Emit(v, $);
        }
    }
    if(input <v, u>){
        Emit(v, u);
    }
}

```

**Pseudocodice 4.23.** Pseudocodice funzione *Reduce* della fase 3 *MapReduce* dell'iterazione Denesest Subgraph per grafi non orientati

```

reduce(String v, Iterator vicini){
    // v: Vertice
    // vicini: Lista vertici vicini
    if  vicini.contains($)
        for each w in vicini{
            emit(v, w)
        }
}

```

#### 4.5.2 Pregel

L'implementazione *Pregel* dell'algoritmo prevede la realizzazione di ogni iterazione in 2 *Superstep*. All'inizio di ogni Superstep il nodo Master decide, in base allo stato degli insiemi  $S$  e  $T$  rilevato tramite sistema di aggregatori, quale funzione *compute* eseguire, quella che controlla il grado dei vertici  $v \in S$  o il grado dei vertici  $v \in T$ .

Il modello *Pregel* prevede che ogni arco del grafo sia associate al vertice *Sorgente* dell'arco. Per poter controllare il grado entrante dei vertici in  $T$  sono svolti 2 ulteriori *Superstep* di inizializzazione, nel primo *Superstep* ogni vertice invia un messaggio con il proprio valore ID ai nodi vicini. Nel *SuperStep* successivo ogni vertice registra ogni messaggio ricevuto come arco entrante nella struttura dati associata al vertice.

Nella figura ?? è rappresentato un esempio di un iterazione dell'algoritmo in *Pregel*.

Ad ogni Superstep la funzione *compute* associata al vertice  $v$  calcola il grado del nodo, entrante o uscente a seconda del caso. Se il valore è minore della *soglia* corrente allora è eliminato il vertice dalla partizione in esame e tutti gli archi.

Nel caso è eliminato un vertice  $v$  dall'insieme  $S$ , vengono eliminati gli archi uscenti da  $v$  e vengono inviati i messaggi ai vertici  $u$  collegati dagli archi eliminati. Nel caso di eliminazione di un arco  $v$  dall'insieme  $T$  allora sono inviati messaggi ai vertici collegati da un arco entrante in  $v$ .

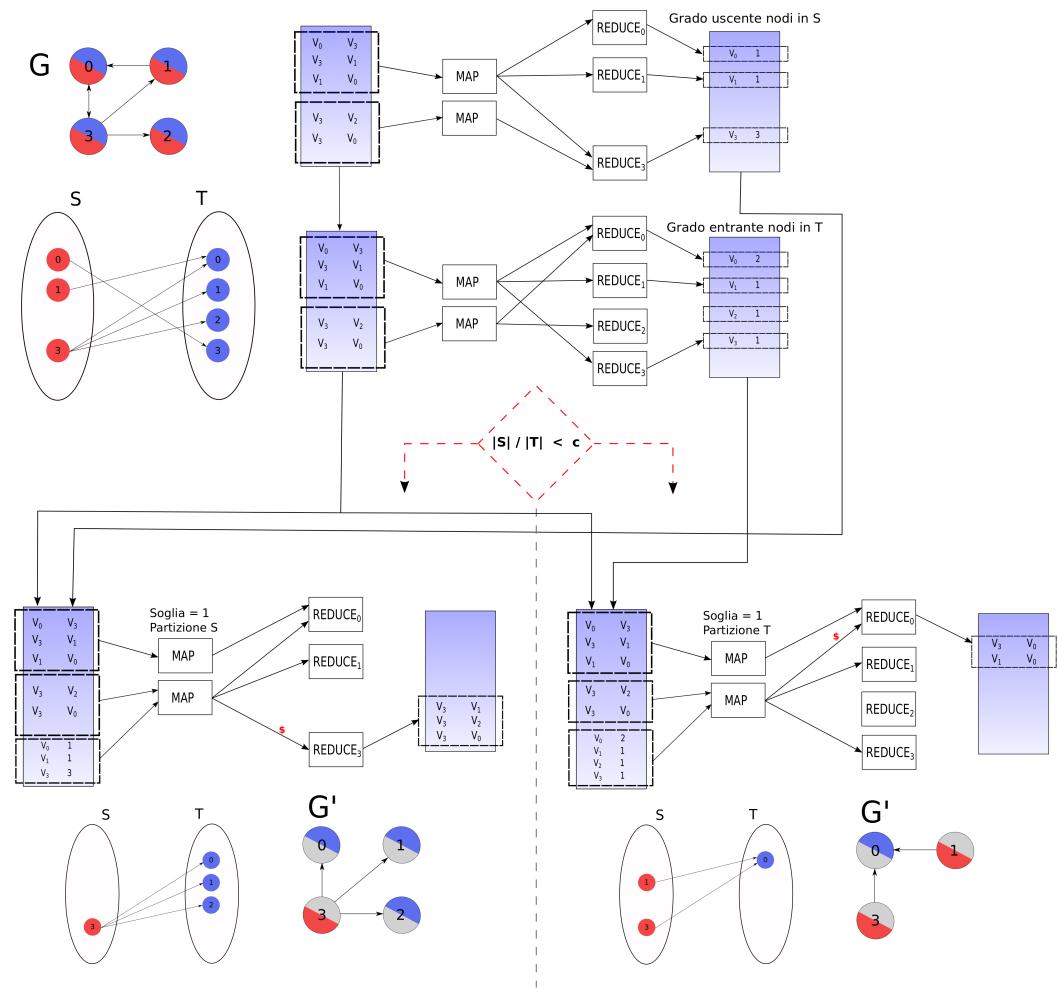
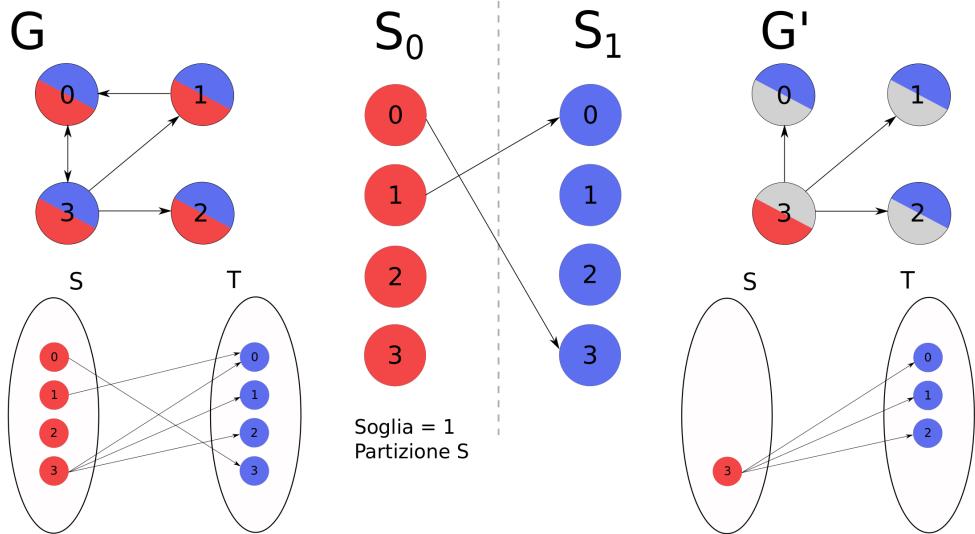


Figura 4.11. Esempio calcolo Densest Subgraph per grafi orientati in MapReduce

I messaggi contengono l'ID del nodo  $v$  eliminato e comunicano al vertice  $u$  che riceve il messaggio, nel secondo *Supertep* dell'iterazione *Pregel*, di eliminare l'arco associato al vertice  $v$ . Nel caso che  $v \in S$  allora il vertice  $u$  elimina l'arco entrante  $(u, v)$ , nel caso  $v \in S$  allora  $u$  elimina l'arco uscente  $(u, v)$ .

Ad ogni iterazione dell'algoritmo è controllata la densità del grafo corrente, nel caso la densità è la maggiore raggiunta dall'algoritmo, viene salvato lo stato del sottografo.

## Insieme S



## Insieme T

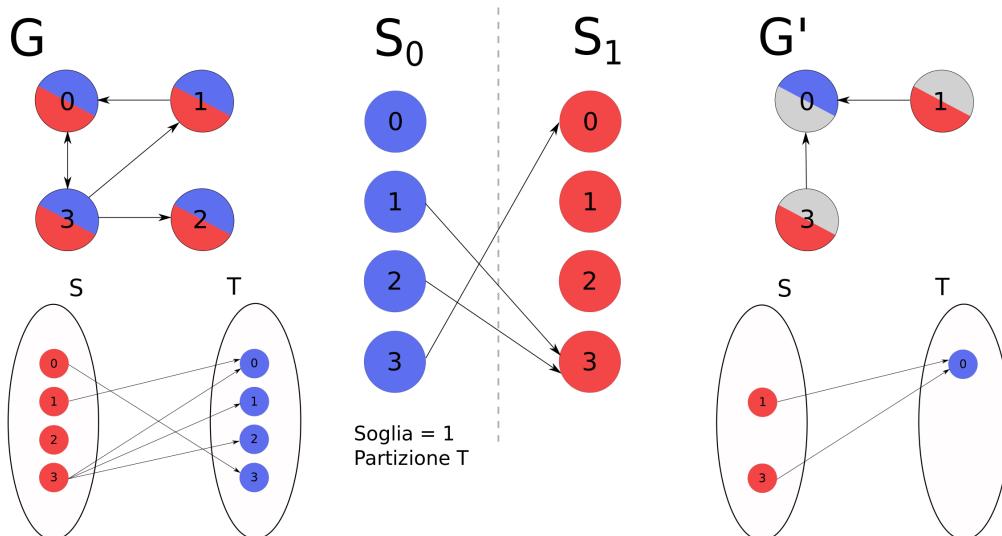


Figura 4.12. Esempio calcolo Densest Subgraph per grafi orientati in Pregel



# Capitolo 5

## Test

### 5.1 Ambiente testing

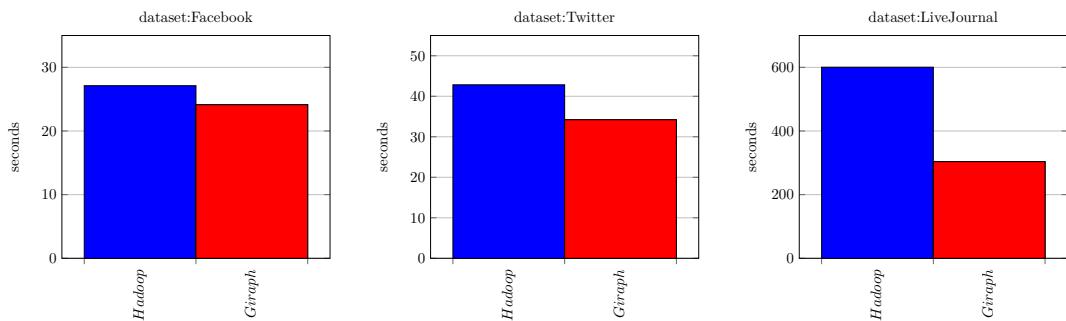
### 5.2 Dataset

### 5.3 Analisi Performance

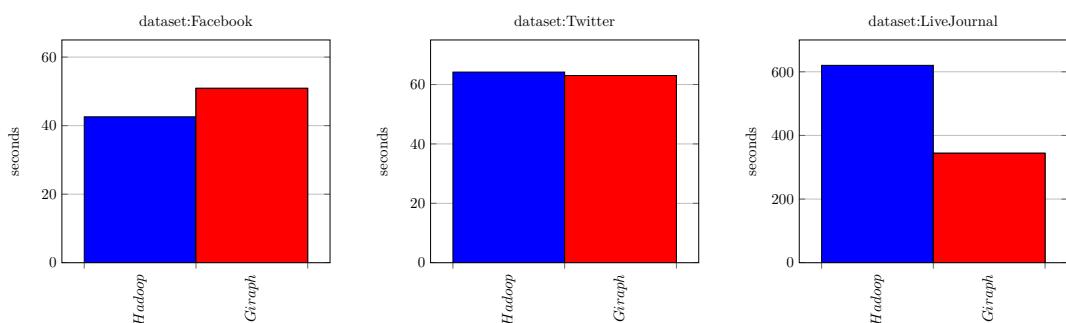
#### 5.3.1 CPU time

##### DegreeComputation

**Tabella 5.1.** DegreeComputation - 8 macchine

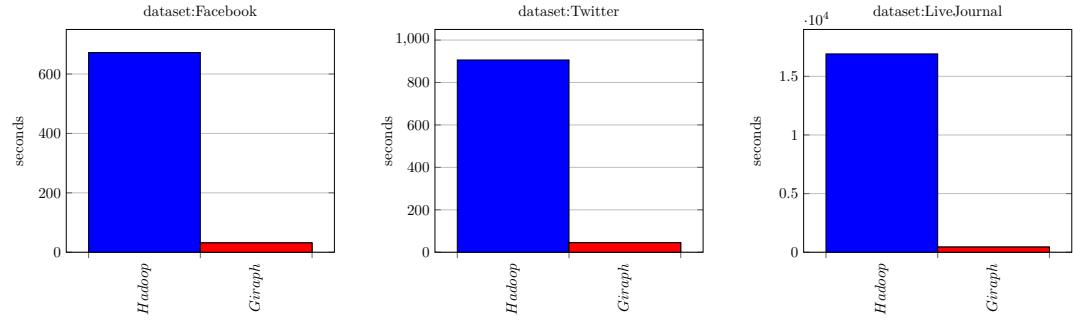


**Tabella 5.2.** DegreeComputation - 16 macchine

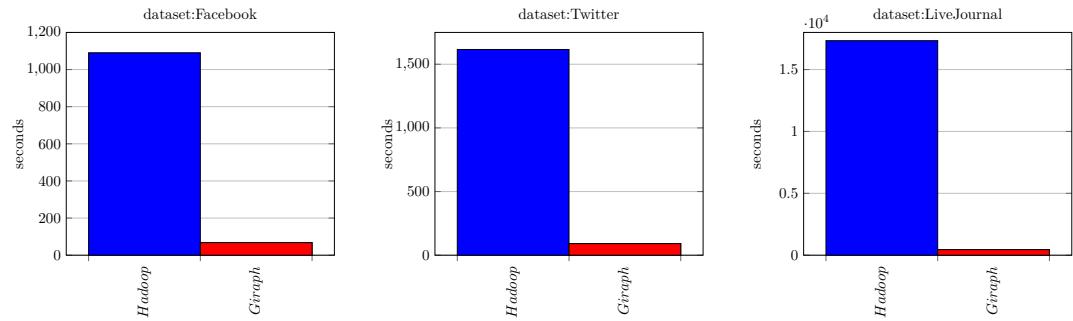


## SSSP

**Tabella 5.3.** SSSP - 8 macchine

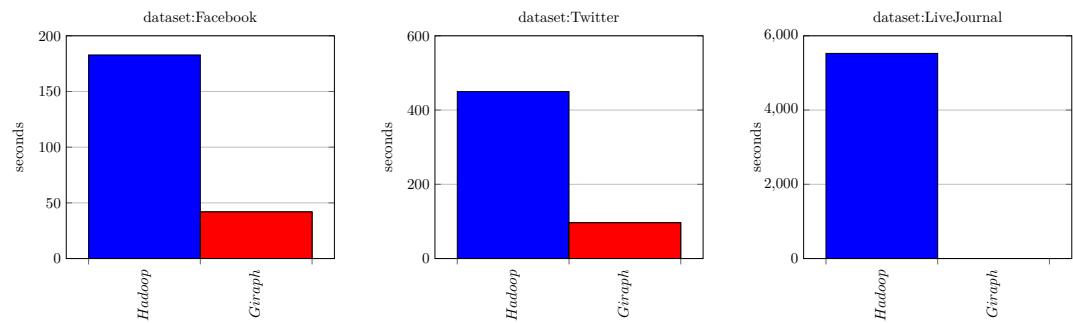


**Tabella 5.4.** SSSP - 16 macchine



## NodeIterator++

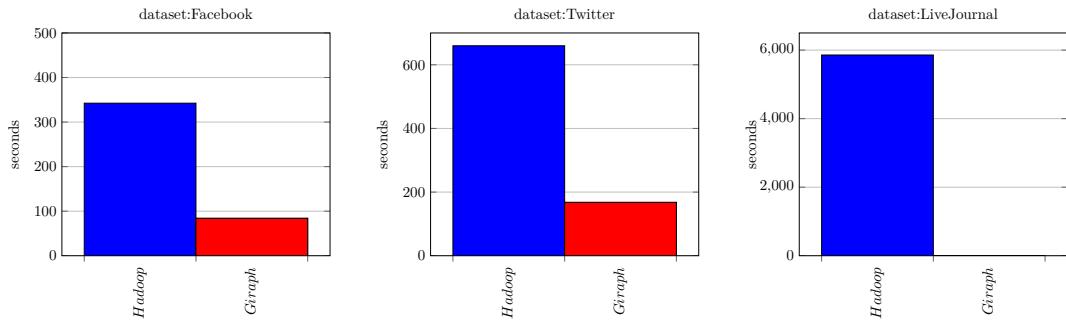
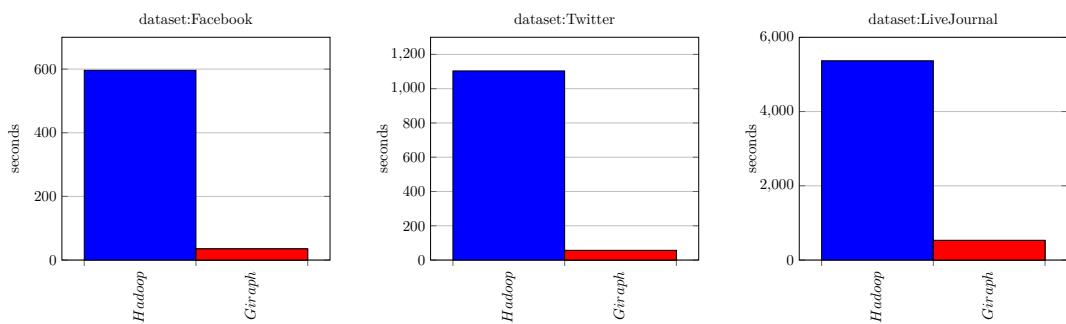
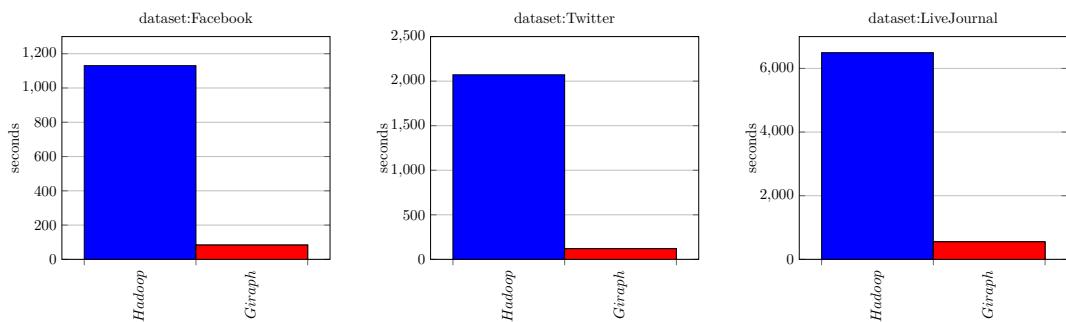
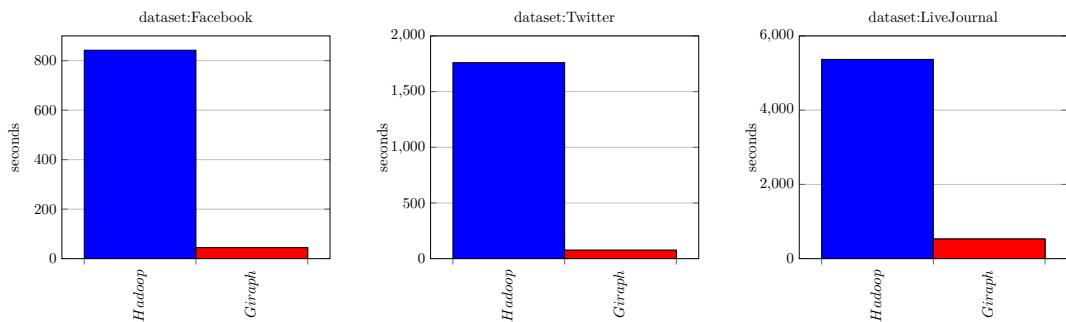
**Tabella 5.5.** NodeIterator++ - 8 macchine

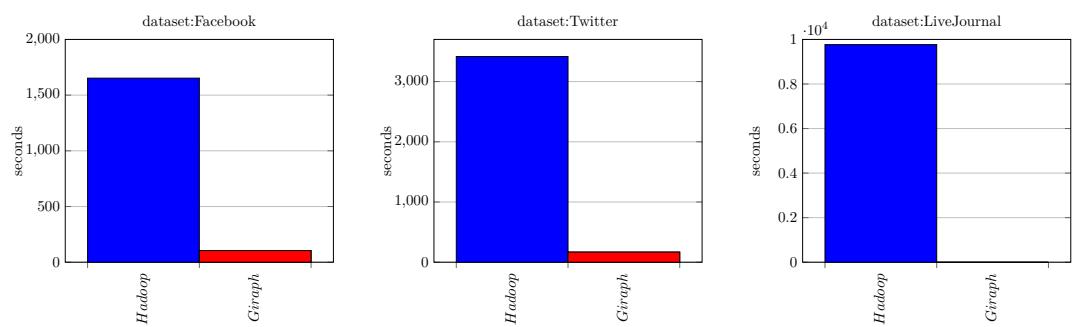


Denesest Subgraph su grafi non orientati

Denesest Subgraph su grafi orientati

### 5.3.2 Utilizzo Memoria

**Tabella 5.6.** NodeIterator++ - 16 macchine**Tabella 5.7.** Denesest Subgraph su grafi non orientati - 8 macchine**Tabella 5.8.** Denesest Subgraph su grafi non orientati - 16 macchine**Tabella 5.9.** Denesest Subgraph su grafi orientati - 8 macchine

**Tabella 5.10.** Denesest Subgraph su grafi orientati - 16 macchine

# Capitolo 6

## Conclusioni



## Elenco delle figure



## Elenco delle tabelle