Calcolo e parallelizzazione della Expected Force

Programmazione su Architetture Parallele

Anno Accademico 2020/2021

Federica De Martin – matricola 139044
Pierfrancesco Bruni – matricola 138388

1 Descrizione del problema

Le misure di centralità come il grado, betweenness, closeness o la centralità dell'autovalore possono identificare i nodi più influenti di una rete, ma raramente sono accurate per quantificare il potere di diffusione dei nodi che non hanno un valore elevato delle metriche citate (nodi non hub). Il potere di diffusione è intuibile, da un punto di vista epidemiologico a tempo continuo, come la distribuzione della forza di infezione generata da ciascun nodo, ovvero la forza con cui può spingere un processo di propagazione al resto della rete. La metrica risultante, chiamata expected force (ExF), quantifica accuratamente tale caratteristica dei nodi in tutti i modelli epidemiologici. L'expected force può essere calcolata in modo indipendente su ogni nodo, rendendola adatta ad un processo di parallelizzazione.

1.1 Definizione della metrica

L'ExF è una proprietà del nodo che deriva dalla topologia della rete locale, ed è formalmente definita come segue.

1. Si consideri una rete con un nodo i infetto e tutti i nodi rimanenti suscettibili. Dopo due eventi di trasmissione (senza recupero) è possibile enumerare tutti i possibili **cluster** $J = 1,...,|J|$ di nodi infetti. Ogni cluster include una possibile combinazione di i più due nodi a distanza uno oppure un nodo a distanza uno ed uno a distanza due. L'enumerazione è su tutti i possibili ordinamenti degli eventi di trasmissione, quindi, due vicini del seme, a e b, formano due cluster: [i → a, i → b] e [i → b, i → a].

2. Si definisce il **grado** di un cluster $\mathrm{d_j}$ come il numero di archi che collegano i nodi all'interno del cluster con quelli all'esterno. In seguito alle due trasmissioni senza recupero, la forza d'infezione di un processo di diffusione seminato dal nodo i è una variabile casuale discreta che assume un valore nell'intervallo $d_1, ..., d_J$.

3. La forza attesa, chiamata appunto ExF, può essere approssimata dall'**entropia** di d_j dopo la sua normalizzazione:

   $ExF(i) = -\sum_{j=J}^{J} \overline{d_\mathrm{j}} log(\overline{d_\mathrm{j}})$ *dove i è il nodo seme e* $\overline{d_\mathrm{k}} = \frac{d_\mathrm{k}}{\sum_J d_\mathrm{j}}$

L'impostazione del numero di trasmissioni pari a due è consigliata ma non necessaria, aumentando tale numero non si aggiungono informazioni rilevanti ed il costo computazionale cresce significativamente.

1.2 Soluzione seriale

Il codice della soluzione seriale del problema affrontato si compone di due parti principali: la prima riguarda la lettura del **grafo diretto** non pesato di input in formato .txt in cui ogni riga del file contiene i due nodi che compongono un arco e la trasformazione dei dati in formato **CSR** (vettori **IC** e **IR**); la seconda parte si occupa di calcolare la metrica ExF su ogni nodo a partire da IC e IR.
Il formato CSR (Compressed Sparse Row) rappresenta una matrice di adiacenza M attraverso tre vettori unidimensionali chiamati *IR*, *IC* e *VAL*. Il vettore *IR* (dimensione pari al numero di nodi più uno 0 in testa) è definito come *IR[0] = 0; IR[i] = IR[i-1] + numero di elementi non nulli nella i-esima riga di M*. Il vettore *IC* (dimensione pari al numero di archi) contiene l'indice di colonna degli elementi di *M*. Il vettore *VAL* (dimensione pari al numero di archi) contiene tutti i valori non nulli di M; poiché nel caso in esame i grafi di input non sono pesati, i valori sono pari a 1 e quindi *VAL* non è stato considerato.

1.2.1 Prima parte

1. Apertura file *.txt* e trasferimento delle due colonne (testa e coda degli archi) in un vettore di coppie (pseudocodice: 2).

2. Ordinamento crescente dei primi elementi delle coppie rispettando la composizione dell'arco.

3. Creazione di un vettore unione di tutti i primi elementi delle coppie seguiti rispettivamente da tutti i secondi elementi.
4. Sostituzione dei nomi dei nodi con il loro indice per ottenere valori nell'intervallo da 1 ad n, con n pari al numero di nodi (pseudocodice: 3):
   (a) creazione di un vettore associativo in cui la chiave è il nome originario del nodo ed il valore è l'indice del nodo (senza ripetizioni) nel vettore unione;
   (b) sostituzione della chiave del nodo con il suo valore nel vettore unione.
5. Per ricavare IC è sufficiente aggiungere uno 0 in testa alla seconda metà del vettore unione dopo le sostituzioni.
6. Per ricavare IR è necessario contare il numero di ripetizioni di ogni nodo nella prima metà del vettore unione e sommarlo al valore del nodo precedente (aggiungendo uno 0 in testa).

1.2.2 Seconda parte

1. Il ciclo *for* applica la funzione *expectedForce()* ad ogni nodo del grafo (pseudocodice:1).
2. I passi della funzione *expectedForce()*(pseudocodice: 5) sono i seguenti:
   (a) Calcolo del numero dei vicini a distanza 1 dal nodo seme sfruttando IR (se non ce ne sono $ExF = 0$).
   (b) Recupero dei vicini del nodo seme scandendo IC e inserimento in un vettore *distOne*.
   (c) Se il vettore *distOne* è vuoto allora ExF=0 (questo succede se un nodo punta solo a se stesso).
   (d) Scandendo il vettore *distOne* con due cicli for annidati, calcolo di tutte le possibili combinazioni a due a due dei nodi a distanza 1 ottenendo ciascuna volta un cluster di tre elementi.
   (e) Per ogni cluster richiamo della funzione *gradoCluster()* (pseudocodice: 4) che calcola il grado del cluster considerato:
      (a) per ogni nodo nel cluster conteggio del numero dei suoi vicini (esclusi quelli già nel cluster);
      (b) il grado totale è ottenuto sommando il numero di vicini dei tre nodi del cluster.
   (f) I gradi dei cluster dei nodi a distanza 1 dal seme, così calcolati, devono essere ripetuti ciascuno due volte (perché si considera l'ordinamento).
   (g) Per ogni nodo in *distOne[]* calcolo dei relativi nodi a distanza 1 (distanza 2 dal nodo seme).
   (h) Creazione dei cluster corrispondenti [nodo seme, nodo a distanza 1 e nodo a distanza 2] e richiamo della funzione *gradoCluster()*.
   (i) Applicazione della formula per il calcolo dell'ExF sul vettore dei gradi dei cluster.

Algorithm 1 Main

```
1: IC[], IR[] ← calcoloICIR(IC, IR)
2: exf ← 0
3: for seed = 1 → IR.size() − 1 do
4:     exf ← expectedForce(IR,IC,seed)
```

Algorithm 2 calcoloICIR

Input: IC, IR
Output: IC, IR

1: $graph\_snap \leftarrow open(\text{"}inputGraph.txt\text{"})$
2: $vect\_paire < int, int > \leftarrow push(graph\_snap(archHead), graph\_snap(archTail))$
3: close("$inputGraph.txt$")
4: sort($vect\_paire$)
5: $vect\_union[] \leftarrow first\_element(vect\_paire) \cup second\_element(vect\_paire)$
6: $output \leftarrow$ **changeVect(vect_union,vect_union.size())**
7: $IC[] \leftarrow vect\_union[\frac{vect\_union.size()}{2}; vect\_union.size()]$
8: $IC[] \leftarrow push\_in\_head(0)$
9: $preIR[] \leftarrow vect\_union[0; \frac{vect\_union.size()}{2}]$
10: $IR[] \leftarrow 0$
11: $j \leftarrow 0$
12: $i \leftarrow 0$
13: $sum \leftarrow 1$
14: **while** $j < preIR.size()$ **&&** $i < preIR.size() - 1$ **do**
15: **if** $preIR[i] \neq$ preIR[i+1] **then**
16: $IR \leftarrow push(sum)$
17: $j \leftarrow j + 1$
18: **else**
19: $sum \leftarrow sum + 1$
20: $i \leftarrow i + 1$
21: **end if**
22: **end while**
23: $IR[] \leftarrow push\_in\_head(0)$
24: **return** $IR[], IC[] = 0$

Algorithm 3 changeVect

Input: vect_union, vect_union.size()
Output: vect_union

```
1: map < int, int > ranks;
2: rank ← 1
3: for index = 0 → vect_union.size() do
4:     element ← vect_union[index]
5:     if ranks[element] == 0 then
6:         ranks[element] ← rank
7:         rank ← rank + 1
8:     end if
9: end for
10: for index = 0 → vect_union.size() do
11:     element ← vect_union[index]
12:     vect_union[index] ← ranks[vect_union[index]]
13: end for
14: return vect_union
```

Algorithm 4 gradoCluster

Input: nodiCluster, IR, IC
Output: grado

```
1: grado ← 0
2: for a = 0 → nodiCluster.size() do
3:     nodo ← nodiCluster[a]
4:     row_value ← IR[nodo] − IR[nodo − 1]
5:     if row_value > 0 then
6:         for k = IR[nodo − 1] + 1 → IR[nodo − 1] + 1 + row_value do
7:             neighbor ← IC[k]
8:             if neighbor ≠ nodiCluster[0]&&neighbor ≠ nodiCluster[1]&&neighbor ≠
   nodiCluster[2] then
9:                 grado ← grado + 1
10:            end if
11:        end for
12:    end if
13: end for
14: return grado
```

Algorithm 5 expectedForce

Input: IR, IC, seed
Output: exf

1: $gradi[]$
2: $exf \leftarrow 0$
3: $totalFI \leftarrow 0$
4: $distOne[]$
5: $dist \leftarrow IR[seed] - IR[seed-1]$
6: $row\_value \leftarrow IR[seed]$
7: **if** $dist == 0$ **then**
8: $exf \leftarrow 0$
9: **else**
10: **for** $k = IR[seed-1]+1 \rightarrow row\_value$ **do**
11: $b\_value \leftarrow IC[k]$
12: **if** $b\_value \neq seed$ **then**
13: $distOne \leftarrow push(b\_value)$
14: **end if**
15: **end for**
16: **if** $empty(distOne)$ **then**
17: $exf \leftarrow 0$
18: **else**
19: **for** $k = 0 \rightarrow distOne.size() - 1$ **do**
20: **for** $j = k+1 \rightarrow distOne.size()$ **do**
21: $nodiCluster \leftarrow push(seed, distOne[k], distOne[j])$
22: $grado \leftarrow$ **gradoCluster(nodiCluster,IR,IC)**
23: $gradi \leftarrow push(grado)$ ▷ Ripetuto 2 volte
24: $totalFI \leftarrow totalFI + grado * 2$
25: **end for**
26: **end for**

▷ Ripeto da riga 19 a riga 26 considerando i cluster formati da (nodo,seme, nodo a distanza 1, nodo a distanza 2) e sommando il grado a totalFI

27: $norm \leftarrow 0$
28: **for** $i = 0 \rightarrow gradi.size()$ **do**
29: **if** $gradi[i] \neq 0$ **then**
30: $norm \leftarrow gradi[i]/totalFI$
31: $exf \leftarrow exf - log(norm) * norm$
32: **end if**
33: **end for**
34: **end if**
35: **end if**
36: **return** exf

2 Parallelizzazione

Di seguito sono riportati i quattro tentativi di parallelizzazione del codice seriale effettuati attraverso CUDA NVIDIA.

2.1 Soluzione parallela 1

2.1.1 Prima parte

Il calcolo dei vettori *IC* e *IR* segue lo stesso procedimento della soluzione seriale (pseudocodice: 2, 3). L'unica differenza risiede nella funzione *calcoloICIR()* dove viene restituito il massimo grado (*gradoMax*) dei nodi del grafo di input, necessario per definire la dimensione massima degli array *distOne[]* e *gradi[]* usati nel codice del kernel.

2.1.2 Seconda parte

1. Il ciclo *for* di Algorithm 1 è stato eliminato e sostituito con il lancio di un kernel (pseudocodice: 6) in modo tale che ogni thread si occupi di calcolare la metrica ExF su un nodo. I vettori *IC* e *IR* sono stati trasformati in *thrust::device_vector* in modo da poter essere passati al kernel attraverso i relativi *thrust::raw_pointer_cast*.

2. È necessario aumentare la dimensione della memoria **heap** ad 1GB per permettere ad ogni thread di inizializzare i due array *distOne[]* e *gradi[]* con dimensione pari al caso peggiore. Nel caso peggiore l'array *distOne[]* ha dimensione *gradoMax*; l'array *gradi[]*, invece, può contenere *gradoMax * gradoMax + gradoMax * (gradoMax - 1)* elementi. È stata presa questa decisione poiché non è possibile inizializzare vettori in un kernel CUDA né utilizzare array dinamici (memorizzati nella shared memory) essendo i due array privati per ogni singolo thread.

3. All'interno della funzione *expectedForce()*(pseudocodice:7) sono stati fatti i seguenti cambiamenti.

   (a) Ogni thread calcola la posizione nell'array *IR* del nodo di cui deve calcolare la metrica.

   (b) Il calcolo procede soltanto se l'indice del nodo considerato non eccede la dimensione di *IR*.

Algorithm 6 Main_parallelo

1: $cudaDeviceSetLimit(cudaLimitMallocHeapSize, 1073741824)$
2: $vect\_h\_IC, vect\_h\_IR$
3: $maxDegree \leftarrow$ **calcoloICIR(vect_h_IC,vect_h_IR)**
4: $n\_IR \leftarrow vect\_h\_IR.size()$
5: $thrust :: device\_vector < int > d\_IR = vect\_h\_IR$
6: $thrust :: device\_vector < int > d\_IC = vect\_h\_IC$
7: $int * IR\_vec = thrust :: raw\_pointer\_cast(d\_IR.data())$
8: $int * IC\_vec = thrust :: raw\_pointer\_cast(d\_IC.data())$
9: $threadsPerBlock = 1024$
10: $blocksPerGrid = (n\_IR + threadsPerBlock - 1)/threadsPerBlock$
11: **expectedForce** $<<< blocksPerGrid, threadsPerBlock >>>$ $(IR\_vec, IC\_vec, n\_IR, maxDegree)$

Algorithm 7 expectedForce_parallelo1

Input: IR, IC, n_IR, gradoMax

```
1: seed ← blockDim.x * blockIdx.x + threadIdx.x
2: if seed < n_IR && seed ≠ 0 then
3:     exf ← 0
4:     totalFI ← 0
5:     row_value ← IR[seed]
6:     indiceGradi ← 0
7:     int * distOne ← new int[gradoMax]
8:     if distOne == NULL then printf("distOnefailed")
9:     end if
10:    dist ← IR[seed] − IR[seed − 1]
11:    if dist == 0 then
12:        exf ← 0
13:    else
14:        indiceDistOne ← 0
15:        for k = IR[seed − 1] + 1 → row_value do
16:            b_value ← IC[k]
17:            if b_value ≠ seed then
18:                distOne[indiceDistOne] ← valB
19:                indiceDistOne ← indiceDistOne + 1
20:            end if
21:        end for
22:        if indiceDistOne == 0 then
23:            exf ← 0
24:        else
25:            int * gradi ← new int[gradoMax * gradoMax + gradoMax * (gradoMax − 1)]
26:            if gradi == NULL then printf("gradifailed")
27:            end if
28:            for k = 0 → indiceDistOne − 1 do
29:                for j = k + 1 → indiceDistOne do
30:                    nodiCluster ← push(seed, distOne[k], distOne[j])
31:                    grado ← gradoCluster(nodiCluster,IR,IC)
32:                    gradi ← push(grado)                          ▷ Ripetuto 2 volte
33:                    totalFI ← totalFI + grado * 2
34:                end for
35:            end for
               ▷ Ripeto da riga 28 a riga 35 considerando i cluster formati da [nodo,seme, nodo a
   distanza 1, nodo a distanza 2] e sommando il grado a totalFI
36:            delete[] distOne
37:            norm ← 0
38:            for i = 0 → indiceGradi do
39:                if gradi[i] ≠ 0 then
40:                    norm ← gradi[i]/totalFI
41:                    exf ← exf − log(norm) * norm
42:                end if
43:            end for
44:            delete[] gradi
45:        end if
46:    end if
47: end if
```

2.2 Soluzione parallela 2

Rispetto alla soluzione parallela 1 in questo caso è stato utilizzato il parallelismo **dinamico**: i cluster formati da nodo seme (definito dal thread della **parent grid**) e da due nodi a distanza 1 e il relativo grado vengono calcolati dai thread di una **child grid**.

1. All'interno della funzione *expectedForce()* è stato fatto il seguente cambiamento: ogni thread lancia una child grid chiamando la funzione *clusterDistOne()* e ogni thread della child grid calcola il grado dei cluster composti da [nodo seme definito dal thread della parent grid, nodo a distanza 1 definito dal thread della child grid, nodo a distanza 1](pseudocodice: 8).

2. Ogni thread della funzione *clusterDistOne()*(pseudocodice: 9) calcola l'indice del nodo a distanza 1 che forma il secondo elemento del cluster considerato. Il calcolo procede soltanto se l'indice del nodo considerato non eccede la dimensione dell'array *distOne[]*. Una volta definito il nodo seme e il nodo a distanza 1, il terzo elemento del cluster viene trovato scandendo l'array *distOne[]* e per ogni cluster viene chiamata la funzione *gradoCluster()*.

Nelle soluzioni in cui è stato utilizzato il parallelismo dinamico il numero di pending child grid è stato impostato a 32.768.

Algorithm 8 expectedForce_parallelo2

Input: IR, IC, n_IR, gradoMax

1: $seed \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2: **if** $seed < n\_IR$ **&&** $seed \neq 0$ **then**
. . . .
3: ▷ Fino a qui il codice è uguale al parallelo 1
4: $int * gradi \leftarrow new\, int[gradoMax * gradoMax + gradoMax * (gradoMax - 1)]$
5: **if** $gradi == NULL$ **then** $printf("gradi failed")$
6: **end if**
7: $threadsPerBlock \leftarrow 32$
8: $blocksPerGrid \leftarrow (indiceDistOne + threadsPerBlock - 1)/threadsPerBlock$
9: **clusterDistOne** $<<<$ $blocksPerGrid, threadsPerBlock$ $>>>$ $(seed, distOne, IR, IC, gradi, indiceGradi, totalFI, indiceDistOne)$
10: $cudaDeviceSynchronize()$
▷ Calcolo il grado dei cluster formati da [nodo seme, nodo a distanza 1, nodo a distanza 2] e sommo il grado a totalFI
11: $delete[]\, distOne$
12: $norm \leftarrow 0$
13: **for** $i = 0 \rightarrow indiceGradi$ **do**
14: **if** $gradi[i] \neq 0$ **then**
15: $norm \leftarrow gradi[i]/totalFI$
16: $exf \leftarrow exf - log(norm) * norm$
17: **end if**
18: **end for**
19: $delete[]\, gradi$
20: **end if**

Algorithm 9 clusterDistOne

Input: seed,distOne, IR, IC, gradi, indiceGradi, totalFI, indiceDistOne

1: $k \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $k < indiceDistOne$ **then**
3: **for** $j = k + 1 \rightarrow indiceDistOne$ **do**
4: $nodiCluster \leftarrow push(seed, distOne[k], distOne[j])$
5: $grado \leftarrow$ **gradoCluster(nodiCluster,IR,IC)**
6: $gradi[indiceGradi + 2 * k] \leftarrow grado$
7: $gradi[indiceGradi + 2 * k + 1] \leftarrow grado$
8: $atomicAdd(totalFI, 2 * grado)$
9: $atomicAdd(indiceGradi, 1)$
10: **end for**
11: **end if**

2.3 Soluzione parallela 3

Nella seguente soluzione anche i cluster formati da [nodo seme, nodo a distanza 1, nodo a distanza 2] sono calcolati dai thread di una child grid.

1. All'interno della funzione *expectedForce()* è stato fatto il seguente cambiamento: ogni thread lancia una child grid chiamando la funzione *clusterDistOneTwo()* e ogni thread della child grid calcola il grado dei cluster composti da [nodo seme definito dal thread della parent grid, nodo a distanza 1 definito dal thread della child grid, nodo a distanza 2] (pseudocodice: 10).

2. Ogni thread della funzione *clusterDistOneTwo()* (pseudocodice: 11) calcola l'indice del nodo a distanza 1 che forma il secondo elemento del cluster considerato. Il calcolo procede soltanto se l'indice del nodo considerato non eccede la dimensione dell'array *distOne[]*. Una volta definito il nodo seme e il nodo a distanza 1, il terzo elemento del cluster viene trovato scandendo i vicini del nodo a distanza 1 e per ogni cluster viene chiamata la funzione *gradoCluster()*.

Algorithm 10 expectedForce_parallelo3

Input: IR, IC, n_IR, gradoMax

1: $seed \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2: **if** $seed < n\_IR$ **&&** $seed \neq 0$ **then**
. . . .
3:
4: $int * gradi \leftarrow new\, int[gradoMax * gradoMax + gradoMax * (gradoMax - 1)]$
5: **if** $gradi == NULL$ **then** $printf("gradi failed")$
6: **end if**
7: $threadsPerBlock \leftarrow 32$
8: $blocksPerGrid \leftarrow (indiceDistOne + threadsPerBlock - 1)/threadsPerBlock$
9: **clusterDistOne** $<<<$ $blocksPerGrid, threadsPerBlock$ $>>>$ $(seed, distOne, IR, IC, gradi, indiceGradi, totalFI, indiceDistOne)$
10: $cudaDeviceSynchronize()$ ▷ Fino a qui il codice è uguale al parallelo 2
11: **clusterDistOneTwo** $<<<$ $blocksPerGrid, threadsPerBlock$ $>>>$ $(seed, distOne, IR, IC, gradi, indiceGradi, totalFI, indiceDistOne)$
12: $cudaDeviceSynchronize()$
. . . . .
13: **end if**
14:

Algorithm 11 clusterDistOneTwo

Input: seed,distOne, IR, IC, gradi, indiceGradi, totalFI, indiceDistOne

$k \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
if $k < indiceDistOne$ **then**
  $nodo \leftarrow distOne[k]$
  $row\_value \leftarrow IR[nodo] - IR[nodo-1]$
  if $row\_value > 0$ **then**
    for $j = IR[nodo-1]+1 \rightarrow IR[nodo-1]+1+row\_value$ **do**
      $valB \leftarrow IC[j]$
      $nodiCluster \leftarrow push(seed, nodo, valB)$
      $grado \leftarrow$ **gradoCluster(nodiCluster,IR,IC)**
      $index \leftarrow atomicAdd(indiceGradi, 1)$
      $gradi[index] \leftarrow grado$
      $atomicAdd(totalFI, grado)$
    end for
  end if

2.4 Soluzione parallela 4

La quarta parallelizzazione teorizzata prevede che per normalizzare l'array *gradi[]* di ogni nodo venga lanciato un kernel e ogni child thread si occupi di normalizzare un elemento dell'array. Ogni elemento normalizzato deve essere poi sommato agli altri per ottenere il valore della *ExF*, quindi deve essere eseguita una **riduzione**. Il problema riscontrato, che non ha permesso la compilazione di questa soluzione, riguarda la funzione *atomicAdd* con parametri double che è disponibile solo a partire da CUDA 8 con supporto hardware nelle GPU SM_6x (Pascal).

3 Risultati sperimentali

3.1 Architettura utilizzata

Il calcolatore utilizzato per i test si basava su sistema operativo Windows 10. Le specifiche della GPU sono riportate in Fig 1.

```
 ---GeneralInformationfordevice0---
Name: NVIDIA GeForce RTX 2060
Compute capability: 7.5
Clock rate: 1680000
Device copy overlap: Enabled
Kernel execution timeout : Enabled
 ---MemoryInformationfordevice0---
Total global mem: 2147024896
Total constant Mem: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
 ---MPInformationfordevice0---
Multiprocessor count: 30
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)
```

Figure 1: Specifiche architettura

3.2 Istanze utilizzate

La soluzione seriale e le tre soluzioni parallele sono state testate su istanze di tipo **SNAP** di medie e grandi dimensioni scaricabili dal sito ufficiale Stanford Large Network Dataset. Sono stati selezionati sei dataset con numero di nodi che varia tra i 6.000 e i 62.000 e con il numero di archi compreso tra 20.000 e 153.000.

Nome del grafo	**N. nodi**	**N. archi**	**Tipolpogia**
p2p-Gnutella08	6.301	20.777	Connessioni fra host in un protocollo di rete
LastFMAsia	8.114	26.013	Relazioni di follow fra utenti del social LastFM
twitch	9.498	153.136	Relazioni di amicizia fra utenti di twitch
Email	10.876	39.994	Scambio di email fra membri di un istituto
deezer	23.281	92.752	Relazioni di amicizia tra utenti di deezer
p2p-Gnutella31	62.586	147.892	Connessioni fra host in un protocollo di rete

3.3 Performance codice

I grafici riportati in Fig 2 e Fig 3 mostrano il confronto tra i tempi di esecuzione delle quattro soluzioni. Sull'asse delle x sono stati riportati i grafi selezionati in base al crescente numero di nodi e sull'asse delle y sono indicati i tempi di esecuzione in millisecondi.

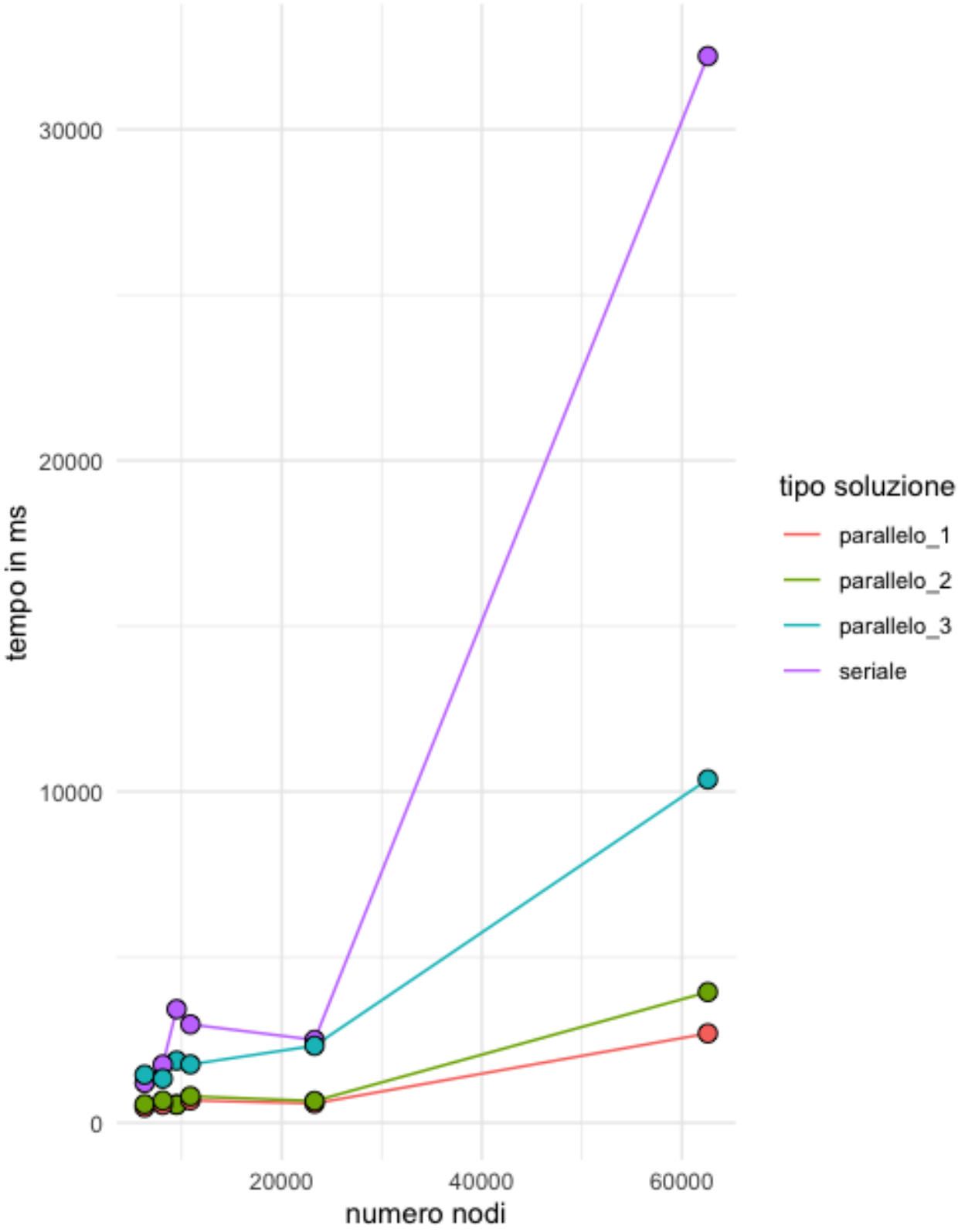

Figure 2: Grafico dei confronti

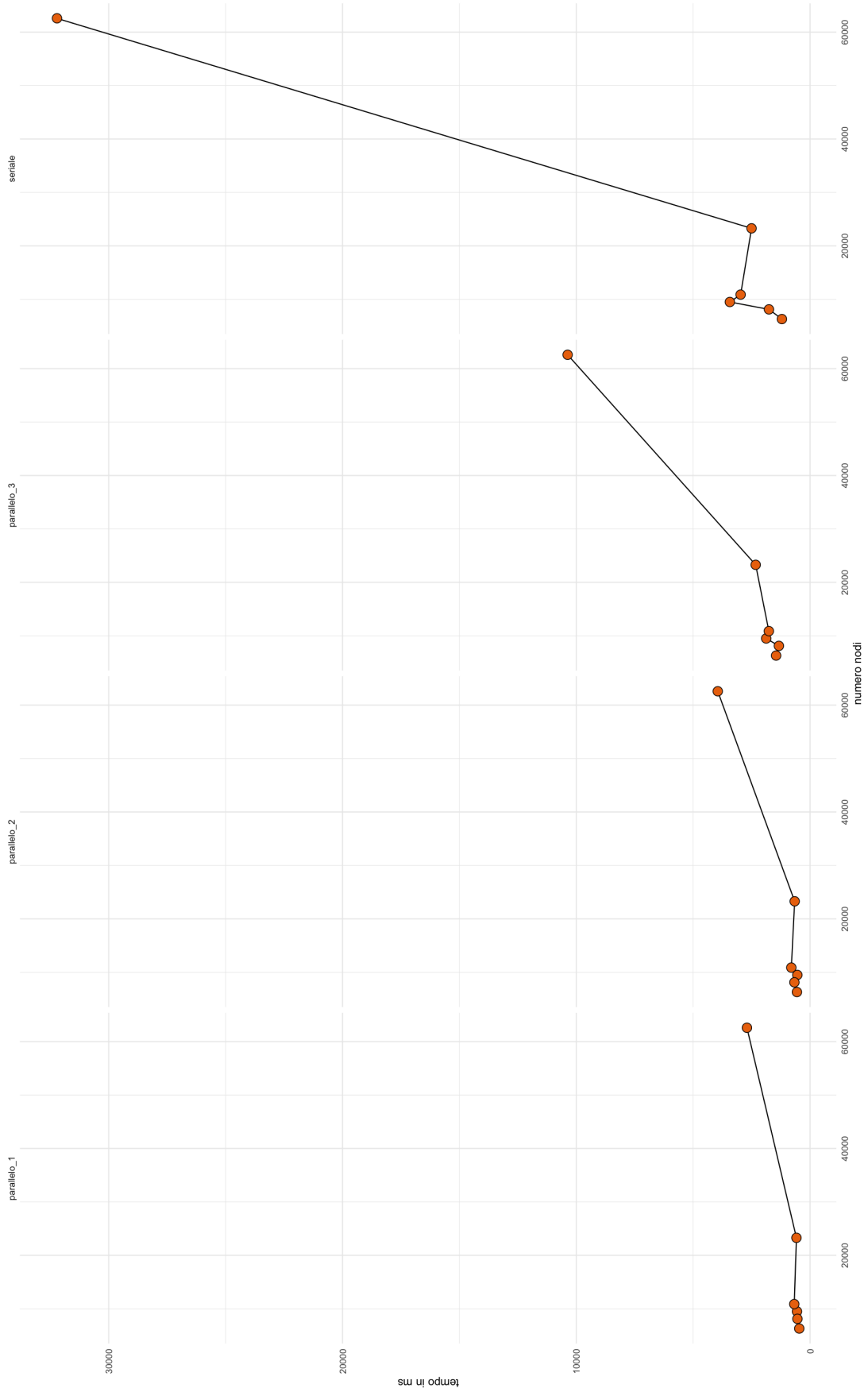

Figure 3: Grafico dei confronti

La tabella sottostante riporta per ogni soluzione la quantità in Megabyte di memoria GPU utilizzata.

	usata	**libera**
parallelo_1	2.060,1875 MB	4.083,3750 MB
parallelo_2	2.380,7500 MB	3.762,8125 MB
parallelo_3	2.380,7500 MB	3.762,8125 MB

3.4 Valutazioni e osservazioni finali

- Le soluzioni parallele proposte supportano soltanto grafi in cui il grado massimo dei nodi non supera il valore di 100; altrimenti sarebbe necessaria una heap di dimensioni maggiori rispetto al massimo consentito dalla GPU utilizzata (anche per grafi con un numero di nodi inferiore a 10.000).
- É possibile notare un netto miglioramento delle prestazioni del codice quando il numero di nodi supera le 20.000 unità nelle soluzioni parallele.
- Tra le diverse soluzioni tentate la prima parallelizzazione è quella con tempistiche migliori, questo potrebbe avere varie motivazioni:
  - ◇ le *atomicAdd*, necessarie per fare in modo che in ogni istante sia un solo thread ad aggiornare il valore dell'indice in cui scrivere il grado del cluster, rendono il flusso delle operazioni simil seriale;
  - ◇ il numero di child grid invocato potrebbe eccedere quello limite e di conseguenza verrebbero messe in attesa.
- La percentuale di memoria GPU utilizzata è quasi sempre la stessa a causa dal settaggio della heap ad 1GB indipendentemente dal tipo di grafo utilizzato.
- Le maggiori difficoltà sono state riscontrate nel momento in cui si è dovuto scegliere la struttura dati necessaria per memorizzare e riutilizzare i gradi dei cluster poiché CUDA non permette l'inizializzazione di vettori nel kernel e con array dinamici la memoria sarebbe stata condivisa fra tutti i thread del blocco (le locazioni sarebbero state scritte e lette in modo inconsistente). Da qui la scelta di memorizzare in modo statico l'array *gradi[]* nella heap con dimensione pari al caso peggiore (ovvero quando un nodo ha *gradoMax* vicini ed ognuno di questi a sua volta ha *gradoMax* vicini.
- Un'altra difficoltà è legata all'aggiornamento degli indici necessari per memorizzare i valori dei gradi dei cluster, per evitare che un thread occupasse locazioni di memoria riservate ad altri thread.
- Ulteriori miglioramenti potrebbero essere raggiunti sfruttando una GPU ancora più performante, capace di supportare le *atomicAdd* della quarta soluzione di parallelizzazione ed un cospicuo aumento della heap utilizzabile.