Implementazione di algoritmi paralleli su GPU per il calcolo d'indici di centralità nell'ambito dell'Analisi delle Reti Sociali

Università di Bologna · Campus di Cesena

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Tirocinante	Riccardo Battistini, 0000873514
Tutor didattico	Moreno Marzolla
Laboratorio	High-Performance Computing
Periodo di svolgimento del tirocinio	01/04/21 - 31/05/21

1. Introduzione

Il tirocinio è stato realizzato nell'ambito del laboratorio di High-Performance Computing per approfondire la programmazione di GPU e acquisire confidenza con il reperimento, la selezione e comprensione di testi di carattere scientifico, oltre che con alcuni concetti e algoritmi fondamentali nel campo dell'Analisi delle Reti Sociali e della Teoria dei Grafi.

2. Tecnologie

Durante il tirocinio è stata sviluppata un'applicazione a linea di comando accelerata dalla GPU per il calcolo di alcune delle principali metriche impiegate nel campo dell'Analisi delle Reti Sociali.

Per implementare l'applicativo sono stati impiegati i linguaggi C, C++ e CUDA C. In particolare il software realizzato impiega il Compute Unified Device Architecture (CUDA) Toolkit 11.2 [16] e gli standard C99 e C++11.

Inoltre durante lo sviluppo dell'applicazione sono stati impiegati The GNU Project Debugger (GDB) e Valgrind. Come sistema per l'automazione dello sviluppo e per effettuare test è stato scelto CMake. Git è stato utilizzato per il controllo di versione.

Per la scrittura della documentazione del progetto sono stati impiegati il linguaggio di markup Asciidoc e Bibtex per la gestione della bibliografia.

Come formati per la memorizzazione dei dati sono stati impiegati Matrix Market, standard per la rappresentazione di matrici sparse [3], e CSV per memorizzare i risultati della valutazione delle prestazioni e della scalabilità.

L'applicazione è stata sviluppata e testata solo sul sistema operativo Ubuntu (18.04).

3. Attività

Inizialmente sono stati consultati gli studi in cui sono state introdotte le metriche che si è scelto d'implementare. In Sezione 3.1 sono brevemente riepilogate.

Successivamente si è sviluppato un programma che fosse in grado di calcolare la Betweenness Centrality prima in modo seriale e in seguito accelerando il calcolo con una GPU. Sono state implementate diverse alternative in base alle diverse strategie che possono essere adottate per suddividere il lavoro fra i thread della GPU. Le diverse strategie sono descritte in Sezione 3.3 e gli algoritmi paralleli sono illustrati in Sezione 3.4.

Prima di procedere è stato necessario riflettere sulla struttura dati principale da impiegare per la rappresentazione della rete e sulla scelta del formato di memorizzazione su disco. In Sezione 3.2 si mostra l'approccio scelto.

Infine è stato effettuato uno studio delle prestazioni e della scalabilità dell'algoritmo implementato su insiemi di dati sintetici e reali. In particolare l'algoritmo su GPU è stato

confrontato con l'algoritmo parallelo della libreria *SNAP* [1]. I risultati sono analizzati in Sezione 3.5.

Come introduzione preliminare e a supporto degli argomenti trattati nel tirocinio sono state consultate diverse fonti [6, 12, 14].

3.1. Misure di Centralità

Le misure di centralità misurano l'importanza di un nodo secondo diverse definizioni. Di seguito si assume che un grafo sia descritto come $G = (V, E)$, dove V è l'insieme dei vertici ed E l'insieme degli archi. Si impiegano n ed m rispettivamente per indicare il numero di vertici e di archi. Infine si considerano solo grafi non diretti e non pesati.

Degree Centrality

Il grado [9] di un nodo rappresenta il numero di collegamenti che ha con gli altri nodi.

Closeness Centrality

La Closeness [9] di un nodo è definita come l'inverso della distanza media tra u e tutti gli altri nodi, ovvero:

$$C(u) = \frac{n-1}{\sum_{v \in V, v \neq u} d(u, v)}$$

Intuitivamente indica quanto rapidamente ci si aspetta che l'informazione prodotta da un nodo u raggiunga il resto della rete [17].

Betweenness Centrality

La Betweenness è una delle più note e diffuse misure di centralità. Si tratta di una misura della proporzione dei cammini minimi di una rete che attraversano uno specifico nodo [8]. Formalmente la *vertex betweenness* è definita nel seguente modo:

$$B(u) = \sum_{v \neq w \neq u \in V} \frac{\sigma_{vw}(u)}{\sigma_{vw}},$$

dove $\sigma_{vw}(u)$ indica il numero di cammini minimi che passano da v a w e σ_{vw} indica il numero di cammini minimi che passano attraverso u.

La Betweenness si può calcolare risolvendo il problema dell'*all-pairs shortest paths* impiegando una versione modificata dell'algoritmo di Floyd-Warshall che segue in tempo $\theta(n^3)$. Brandes [4] ha proposto un algoritmo migliore che esegue in tempo $\theta(nm)$. In Sezione 3.3 saranno discusse diverse possibili implementazioni parallele su GPU basate su quest'ultimo algoritmo.

La Betweenness si presta a diverse interpretazioni. Può esprimere la capacità di un nodo di controllare, distorcere, inibire o bloccare il flusso d'informazioni in una rete oppure può indicare quanto un nodo possa unire due o più gruppi di nodi e favorire lo scambio d'informazioni. Questa metrica è di particolare interesse anche perché è alla base di algoritmi

più complessi come la *Community Detection* e il suo calcolo efficace in parallelo non è facile da realizzare.

3.2. Rappresentazione di un grafo

Un grafo generalmente è rappresentato in due modi: come collezione di liste di adiacenza o come una matrice di adiacenza [14 p. 20].

Dato che per il calcolo delle misure di centralità d'interesse si impiegano degli algoritmi che non richiedono l'inserimento o la rimozione di vertici dai grafi in analisi e che sono necessarie frequenti operazioni di ricerca per determinare se c'è un arco che collega due vertici, si preferisce non impiegare le liste di adiacenza.

In particolare la scelta della matrice di adiacenza permette di definire uno schema di accesso ai dati più efficiente per la cache. Ciò permette di parallelizzare efficacemente e in modo più intuitivo gli algoritmi su grafi.

Dato che i grafi che si incontrano nell'ambito dell'Analisi delle Reti Sociali sono sparsi [2 p. 13], ovvero $|E| \ll |V|^2$, e di grandi dimensioni (da centinaia di migliaia a milioni di vertici e in crescita) il costo $\theta(n^2)$ in termini di occupazione di memoria di una matrice di adiacenza non è sostenibile. Perciò si sfruttano le caratteristiche strutturali di questi grafi rappresentandoli come matrici sparse, ovvero come matrici in cui gli elementi nulli sono omessi. In questo modo è possibile ottenere un notevole risparmio in termini di occupazione di memoria in funzione del grado di sparsità del grafo.

Alcuni dei formati di memorizzazione di matrici sparse più noti sono *COOrdinate Format* (COO) e *Compressed Sparse Row* (CSR). Il formato COO è impiegato dallo standard Matrix Market scelto per la memorizzazione su disco. Per memorizzare la matrice in memoria principale si è scelto d'impiegare CSR. Il vantaggio del formato CSR sta nell'essere molto simile alla rappresentazione di un grafo tramite liste di adiacenza ma nel non avere i costi aggiuntivi legati all'uso dei puntatori e nell'essere più efficiente nell'uso della cache. Un esempio di rappresentazione di un grafo in formato CSR si ha in Figura 1.

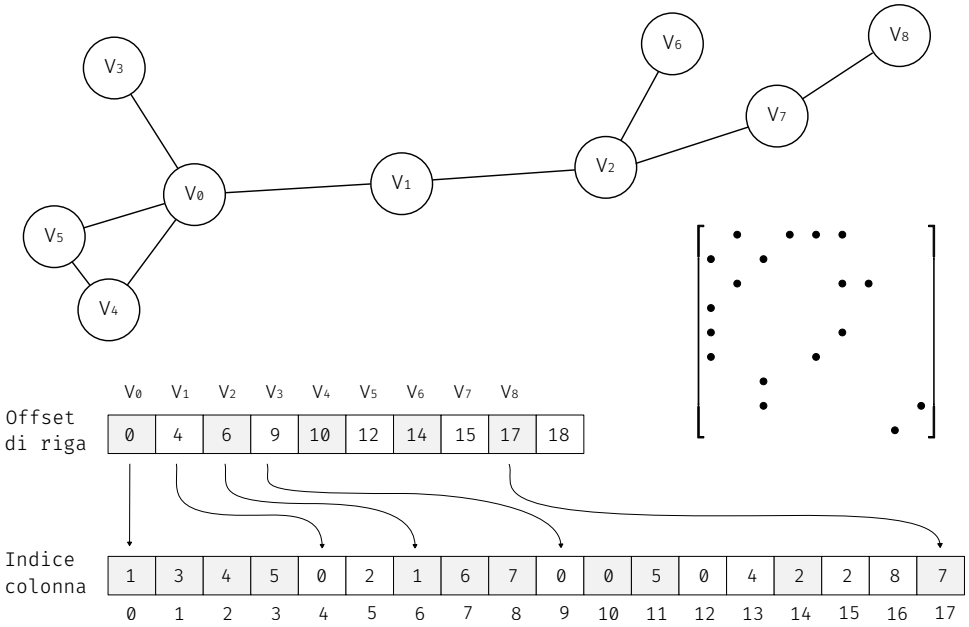

Figura 1. Esempio di rappresentazione di un grafo tramite il formato CSR

3.3. Tecniche di parallelizzazione su GPU

Sia gli algoritmi per il calcolo della Closeness Centrality che della Betweenness Centrality richiedono il computo dei cammini minimi tra due vertici del grafo. Nel caso di grafi non diretti e non pesati si impiega una visita in ampiezza. Questa prima fase è detta anche di propagazione in avanti. Inoltre per la Betweenness Centrality è necessaria una seconda fase, detta anche di propagazione all'indietro, in cui si effettua l'accumulazione delle dipendenze sfruttando l'equazione ricorsiva proposta da Brandes [4]:

$$\delta_{s*}(v) = \sum_{w \in S(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s*}(w))$$

dove $\delta_{s*}(v)$ indica la dipendenza di v rispetto al nodo sorgente s e $S(v)$ rappresenta i successori w di v.

Sia le visite in ampiezza che l'accumulazione delle dipendenze possono essere eseguite in parallelo. Su una GPU Nvidia ciò si ottiene distribuendo un gruppo di vertici a ogni *Streaming Multiprocessor* [16 p. 107]. In questo modo si realizza un parallelismo a grana grossa.

Inoltre, in ciascuno Streaming Multiprocessor, si può strutturare l'attraversamento del grafo per livelli di profondità in modo che sia sincrono. In altri termini a ogni iterazione si ha una frontiera di vertici o archi attivi che devono essere processati. Quando è stata completata l'elaborazione dei vertici o degli archi di una frontiera allora si può avanzare al livello successivo di profondità. In questo modo tutti i thread appartenenti alla frontiera corrente lavorano concorrentemente nell'esecuzione di un'unica visita in ampiezza o nell'accumulazione delle dipendenze sincronizzata realizzando un parallelismo a grana fine.

Esistono diverse strategie per distribuire l'elaborazione dei vertici o degli archi di una frontiera fra i thread. Jia et al. [11] hanno confrontato due delle strategie che si possono

impiegare, ovvero gli approcci *vertex parallel* ed *edge parallel*.

Nel primo caso si assegna un thread a ogni vertice del grafo e ciascun thread attraversa tutti gli archi del vertice. Ciò significa che, come si osserva nell' Algoritmo 1, ciascun thread ha un carico di lavoro che dipende dal numero di vicini del vertice a cui è assegnato. Perciò in uno stesso *warp* [16 p. 107], ci saranno thread corrispondenti a vertici di grado basso che devono attendere thread assegnati a vertici di grado elevato, in entrambe le fasi dell'algoritmo. Perciò l'approccio vertex parallel non permette di effettuare una parallelizzazione efficiente dato che è soggetto a sbilanciamento del carico. In particolare non è adatto all'applicazione nelle reti Scale-free, dove la varianza del grado dei vertici è alta e segue una legge di potenza.

In alternativa si può assegnare un thread a ogni arco del grafo. Come si può vedere nell' Algoritmo 2, i thread attivi sono solo quelli assegnati ad archi incidenti sulla frontiera. Con questo approccio si può bilanciare il carico di lavoro in quanto per ogni arco o si effettua un aggiornamento o non si eseguono operazioni. Il principale limite sta nel notevole consumo di memoria dovuto al numero di letture richiesto per identificare i vertici della frontiera e al numero di archi molto superiore al numero di vertici.

Per l'analisi di grafi di tipo Scale-free o Small World Jia et al. [11] hanno dimostrato che l'approccio edge parallel è più efficace di vertex parallel in quanto si ha un migliore bilanciamento del carico. In particolare rispetto all'approccio vertex parallel si ha un throughput maggiore e una più bassa divergenza dei thread. Ciò può essere osservato anche in Sezione 3.5.

Algoritmo 1. Pseudocodice della visita in ampiezza applicando la tecnica vertex parallel

```
1   for v in G.V on wavefront in parallel do
2       for w in G.adjacentVertices(v) do
3           if w.d = -1 then
4               label w.d as w.d + 1
5           end
6           if w.d = d + 1 then
7               fetch_and_add(w.σ, v.σ)
8           end
9       end
10  end
```

Algoritmo 2. Pseudocodice della visita in ampiezza applicando la tecnica edge parallel

```
1   for e in G.E incident to wavefront in parallel do
2       if w.d = -1 then
3           label w.d as w.d + 1
4       end
5       if w.d = d + 1 then
6           fetch_and_add(w.σ, v.σ)
7       end
8   end
```

McLaughlin e Bader [15] introducono una terza strategia, un approccio *work efficient*, che permette di evitare di assegnare inutilmente thread a vertici o archi che non fanno parte della frontiera a un dato livello di profondità. In questo approccio si assegnano i thread ai vertici che sono adiacenti ai vertici della frontiera. Come mostrato nell'Algoritmo 3, ciò si realizza impiegando due code per l'attraversamento del grafo, una per i vertici della frontiera corrente e una per i vertici della frontiera successiva, e uno stack. Inoltre si impiega un vettore di supporto, `ends`, per annotare il livello di profondità in cui si trova ciascun vertice. In questo modo è possibile determinare immediatamente quali sono i vertici della frontiera corrente e assegnare solo i thread necessari durante la fase di accumulazione. Per maggiori informazioni riguardo l'implementazione si faccia riferimento allo studio di McLaughlin e Bader [15].

L'approccio work efficient è particolarmente efficace in grafi con un diametro elevato perché in genere hanno una distribuzione del grado più uniforme. In grafi Scale-free o Small World lo sbilanciamento del carico implica che le prestazioni di questo approccio sono migliori di vertex parallel ma non di edge parallel. Ciò si osserva anche in Sezione 3.5.

Algoritmo 3. Pseudocodice della visita in ampiezza applicando la tecnica work efficient

```
1   while True do
2       for v in Q_curr do in parallel
3           for w in G.adjacentVertices(v) do
4               if compare_and_swap(d[w], -1, d[v] + 1) == -1 then
5                   label t as fetch_and_add(Q_next_len, 1)
6                   label w as Q_next[t]
7               end
8               if d[w] = d[v] + 1 then
9                   fetch_and_add(σ[w] , σ[v])
10              end
11          end
12      end
13      barrier()
14      if Q_next_len = 0 then
15          label depth as d[S[S_len - 1]] - 1
16          break;
17      else
18          for tid in 0 ... Q_next_len - 1 do in parallel
19              label Q_curr[tid] as Q_next[tid]
20              label S[tid + S_len] as Q_next[tid]
21          end
22          barrier()
23          label ends[ends_len] as ends[ends_len - 1] + Q_next_len
24          label ends_len as ends_len + 1
25          label Q_curr_len as Q_next_len
26          label S_len as S_len + Q_next_len
27          label Q_next_len as 0
28          barrier()
29      end
30  end
```

3.4. Algoritmo parallelo su GPU

Sono state implementate tre varianti dell'algoritmo per il calcolo della Betweenness Centrality. Ciascuna impiega una dalle strategie viste in Sezione 3.3. In particolare ciascuna strategia impiega per la prima fase dell'algoritmo lo pseudocodice illustrato in Algoritmo 1, Algoritmo 2 e Algoritmo 3.

In Algoritmo 4 si realizza l'accumulazione delle dipendenze impiegando lo stesso metodo di Madduri et al. [13] esposto da McLaughlin e Bader [15]. Questo approccio consiste nell'effettuare l'accumulazione controllando i successori e non i predecessori di ciascun vertice e permette di eliminare la concorrenza nell'aggiornamento delle dipendenze di ciascun vertice. In Algoritmo 5 e Algoritmo 6 sono mostrati rispettivamente l'accumulazione delle dipendenze per la tecnica vertex parallel ed edge parallel. Si osserva che nella tecnica vertex parallel questa fase può essere svolta senza impiegare operazioni atomiche mentre in edge parallel è necessario in quanto potrebbero esserci più thread che processano ciascuno un arco con un estremo in comune concorrentemente.

Inoltre è stato implementato un algoritmo per il calcolo della Closeness Centrality che impiega l'approccio edge parallel.

Algoritmo 4. Pseudocodice dell'accumulazione delle dipendenze applicando la tecnica work efficient

```
1   while depth > 0 do
2       for tid in ends[depth] ... ends[depth + 1] do in parallel
3           label w as S[tid]
4           label dsw as 0
5           label sw as σ[w]
6           for v in G.adjacentVertices(w)
7               if d[v] = d[w] + 1 then
8                   label dsw as dsw + (sw * (1 + δ[v])) / σ[v]
9               end
10          end
11          label δ[w] as dsw
12      end
13      barrier()
14      label depth as depth + 1
15  end
```

Algoritmo 5. Pseudocodice dell'accumulazione delle dipendenze applicando la tecnica vertex parallel

```
1  for v in G.V on wavefront in parallel do
2      for w in G.adjacentVertices(v) do
3          if w.d = v.d + 1 then
4              if w.σ != 0 then
5                  label v.δ as (1 + v.δ) * (v.σ / w.σ)
6              end
7          end
8      end
9  end
```

Algoritmo 6. Pseudocodice dell'accumulazione delle dipendenze applicando la tecnica edge parallel

```
1  for e in G.E incident to wavefront in parallel do
2      if w.d = d + 1 then
3          if w.σ != 0 then
4              fetch_and_add(v.δ, (1 + v.δ) * (v.σ / w.σ))
5          end
6      end
7  end
```

3.5. Studio di scalabilità e throughput

Lo studio di scalabilità e throughput è stato effettuato impiegando una CPU Intel Core i7-10700 con frequenza di funzionamento pari a 2.9 Ghz, una cache di 16 Mb e 16 Gb di DRAM, con Hyper-Threading abilitato. La GPU è una Quadro P620 con quattro Streaming Multiprocessor e clock di base di 2505 Mhz. La memoria GDDR5 a disposizione è pari a due Gb e la *compute capability* è 6.1 (architettura Pascal). I compilatori impiegati sono nvcc 11.2.142 e GCC 10.2.1.

Tabella 1. Proprietà generali dei dataset considerando la più grande componente connessa ed escludendo i cicli

Grafo	Vertici	Archi	Grado massimo	Diametro	Descrizione
rnd-sf	977	1 227	818	6	Grafo di tipo Scale-free
rnd-sw	3 000	2 082	8	22	Grafo di tipo Small World
rnd-er	3 000	900 000	357	2	Grafo casuale Erdös-Rényi
ca-GrQc	4 158	26 844	81	17	Rete di collaborazioni
USpowerGrid	4 941	13 188	19	46	Rete elettrica degli USA
Erdos02	5 534	16 944	507	4	Rete di collaborazioni
ca-HepTh	8 638	49 612	65	17	Rete di collaborazioni
ca-HepPh	11 204	235 238	491	13	Rete di collaborazioni
delaunay_n14	16 384	98 244	16	65	Triangolazione casuale
ca-AstroPh	17 903	393 944	504	14	Rete di collaborazioni
as-22july06	22 963	96 872	2390	11	Topologia d'Internet

I dataset per effettuare i test sono reperibili nella *Sparse Matrix Collection* dell'Università della Florida [7]. In Tabella 1 sono mostrate alcune delle principali proprietà delle reti in analisi.

Per la generazione casuale di grafi secondo i modelli di Erdös-Rényi, Watts-Strogatz e Barabási-Albert [12 p. 7] sono state impiegate le funzioni rese disponibili dalla *Boost Graph Library* [18].

Per il calcolo dei tempi di esecuzione della versione seriale e della versione parallela su CPU a memoria condivisa è stato impiegato l'algoritmo messo a disposizione da SNAP.

Per poter gestire anche grafi non connessi si è scelto di considerare la componente connessa più grande di ciascun grafo in analisi. Per l'estrazione del relativo sottografo sono state utilizzate le funzioni SpGEMM a partire dall'algoritmo di Gustavson [10] e SpRef [5].

Il throughput è misurato impiegando una metrica detta *Traversed Edges Per Second*, o *TEPS*. I TEPS sono stati calcolati nel seguente modo:

$$\text{TEPS} = \frac{2 \cdot n \cdot m}{T_{gpu}}$$

Il calcolo è stato effettuato considerando che durante l'esecuzione della fase di propagazione in avanti degli algoritmi su GPU si effettua una visita in ampiezza a partire da ciascun vertice del grafo. Ciò implica l'attraversamento di tutti gli archi del grafo a partire da ciascun vertice. Nella seconda fase si esegue una propagazione all'indietro che comporta l'attraversamento dello stesso numero di archi della prima fase.

L'algoritmo su GPU è stato lanciato con quattro blocchi, uno per ogni Streaming Multiprocessor, da 1024 thread ciascuno. Per misurare il tempo di esecuzione, sia su CPU che GPU, non si include il tempo impiegato per la lettura del grafo da disco né si tiene conto dell'estrazione della più grande componente connessa. Nel caso dell'algoritmo su GPU è incluso il tempo richiesto per le comunicazioni tra host e device e per la gestione della memoria sul device.

Tabella 2. Confronto in termini di speedup dell'algoritmo parallelo su CPU della libreria SNAP lanciato con 16 thread e delle strategie vertex parallel, edge parallel e work efficient per la distribuzione del lavoro fra i thread su GPU

Grafo	$Speedup_{cpu\_par}$	$Speedup_{vtx\_par}$	$Speedup_{edg\_par}$	$Speedup_{w\_eff}$
rnd-sf	1.50	0.03	2.00	0.04
rnd-sw	3.45	0.59	0.94	1.10
rnd-er	6.68	0.59	1.09	0.74
ca-GrQc	6.97	0.49	1.62	0.85
USpowerGrid	7.03	0.49	0.98	0.69
Erdos02	0.96	0.85	11.12	1.14
ca-HepTh	2.00	1.65	3.40	3.38
ca-HepPh	2.22	1.16	3.41	2.89
delaunay_n14	1.62	1.19	0.82	3.78
ca-AstroPh	2.45	1.59	3.43	3.34
as-22july06	1.09	0.60	5.09	0.76

In Tabella 2 si osserva che lo speedup dei diversi algoritmi varia significativamente a seconda del tipo di grafo considerato. A partire dai dati ottenuti si effettuano le seguenti considerazioni:

- nei grafi in cui il grado massimo è maggiore, come in `rnd-sf`, in `Erdos02` e `as-22july06`, la distribuzione del grado è più ampia e la tecnica vertex parallel ha le prestazioni peggiori. Ciò accade perché questo approccio soffre di uno sbilanciamento del carico che è tanto più pronunciato quanto più può variare il numero di vicini di un vertice.
- Nei grafi in cui in cui il diametro è maggiore e il grado massimo è minore, come in `delaunay_n14`, le prestazioni dell'approccio vertex parallel migliorano al punto da superare l'approccio edge parallel grazie alla distribuzione del grado più uniforme.
- Nei grafi di dimensioni minori i costi in termini di allocazione di risorse per eseguire gli algoritmi su GPU rendono l'algoritmo parallelo su CPU in ogni caso più efficace. Ciò si nota in particolare in `rnd-er`, `rnd-sw`, `ca-GrQc` e `USpowerGrid`. Quest'osservazione è valida nella maggior parte dei casi anche rispetto all'algoritmo seriale su CPU.
- Nei grafi di dimensioni maggiori le prestazioni dell'algoritmo parallelo su CPU calano significativamente. Su GPU si hanno due dei più grandi speedup per i grafi di dimensioni maggiori, `delaunay_n14` e `as-22july06`.
- L'approccio work efficient non è mai più efficace di edge parallel nel caso di grafi che rappresentano reti sociali. Nel migliore dei casi ha prestazioni comparabili a edge parallel, come in `ca-HepPh` e `ca-AstroPh`. Tuttavia in grafi con diametro e dimensioni maggiori la tecnica work efficient determina uno speedup che non si osserva negli altri approcci.
- L'approccio edge parallel è il più adatto se applicato a reti sociali e in generale a reti con diametro relativamente ridotto, di tipo Scale-free o Small World. In caso contrario si hanno prestazioni anche peggiori della versione seriale su CPU.

Queste osservazioni possono essere verificate anche osservando il tempo di esecuzione e il throughput in termini di Milioni di TEPS mostrati in Tabella 3 e Tabella 4.

Tabella 3. Tempo di esecuzione degli algoritmi per il calcolo della Betweenness Centrality su CPU con singolo thread, su CPU con 16 thread e su GPU con le tecniche vertex parallel, edge parallel e work efficient, in Millisecondi

Grafo	**T_{cpu_par}**	**T_{vtx_par}**	**T_{edg_par}**	**T_{w_eff}**
rnd-sf	12	583	9	429
rnd-sw	86	507	317	270
rnd-er	464	5 264	2 848	4 217
ca-GrQc	112	1 607	483	923
USpowerGrid	114	1 644	814	1 158
Erdos02	2 216	2 507	192	1 875
ca-HepTh	4 673	5 679	2 752	2 767
ca-HepPh	17 600	33 860	11 470	13 550
delaunay_n14	18 990	25 880	37 290	8 106
ca-AstroPh	46 400	71 590	33 110	34 010
as-22july06	46 240	84 120	9 929	66 400

Tabella 4. Confronto in termini di throughput delle strategie vertex parallel, edge parallel e work efficient per la distribuzione del lavoro fra i thread su GPU, in Milioni di TEPS

Grafo	**$MTEPS_{vtx\_par}$**	**$MTEPS_{edg\_par}$**	**$MTEPS_{w\_eff}$**
rnd-sf	4.1	266.4	5.6
rnd-sw	24.6	39.4	46.3
rnd-er	1025.8	1896.1	1280.5
ca-GrQc	138.9	462.2	241.9
USpowerGrid	79.3	160.1	112.5
Erdos02	74.8	976.8	100.0
ca-HepTh	150.9	311.4	309.8
ca-HepPh	155.7	459.6	389.0
delaunay_n14	124.4	86.3	397.1
ca-AstroPh	197.0	426.0	414.7
as-22july06	52.9	448.1	67.0

4. Conclusioni

Nel corso del tirocinio ho avuto la possibilità di apprendere alcuni concetti di base sia nel campo dell'Analisi delle Reti Sociali che della Teoria dei Grafi. Inoltre ho impiegato il parallelismo massivo offerto dalle GPU sperimentando diversi approcci alternativi per il calcolo della Betweenness Centrality. In particolare è stato possibile accelerare algoritmi su grafi con successo impiegando una GPU nonostante difficoltà come l'irregolarità della struttura di un grafo e il fatto che il calcolo sia guidato dai dati [12 p. 13]. In conclusione ho acquisito maggiore esperienza nella programmazione in C/C++ e nell'utilizzo di diversi strumenti come CMake, GNU Make, Git e GDB per automatizzare la compilazione del codice, per effettuare test, per il controllo di versione e la verifica della correttezza del programma.

Il software sviluppato durante il tirocinio è liberamente disponibile in un repository.

5. Bibliografia

[1] D. A. Bader e K. Madduri, «SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks», in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, 2008, pagg. 1–12, doi: 10/cnwsfr.

[2] A.-L. Barabási e M. Pósfai, *Network science*. Cambridge: Cambridge University Press, 2016.

[3] R. Boisvert, R. Pozo, e K. Remington, «The Matrix Market Exchange Formats: Initial Design», *NISTIR*, vol. 5935, dic. 1996.

[4] U. Brandes, «A faster algorithm for betweenness centrality*», *The Journal of Mathematical Sociology*, vol. 25, n. 2, pagg. 163–177, giu. 2001, doi: 10/fn726f.

[5] A. Buluc e J. Gilbert, «Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments», *SIAM Journal on Scientific Computing*, vol. 34, n. 4, pagg. C170–C191, gen. 2012, doi: 10.1137/110848244.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[7] T. A. Davis e Y. Hu, «The university of Florida sparse matrix collection», *ACM Transactions on Mathematical Software*, vol. 38, n. 1, pagg. 1:1–1:25, dic. 2011, doi: 10.1145/2049662.2049663.

[8] L. C. Freeman, «A Set of Measures of Centrality Based on Betweenness», *Sociometry*, vol. 40, n. 1, pag. 35, mar. 1977, doi: 10/btcpw5.

[9] L. C. Freeman, «Centrality in social networks conceptual clarification», *Social Networks*, vol. 1, n. 3, pagg. 215–239, gen. 1978, doi: 10/bx3m36.

[10] F. G. Gustavson, «Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition», *ACM Transactions on Mathematical Software*, vol. 4, n. 3, pagg. 250–269, set. 1978, doi: 10.1145/355791.355796.

[11] Y. Jia, V. Lu, J. Hoberock, M. Garland, e J. C. Hart, «Chapter 2 - Edge v. Node Parallelism for Graph Centrality Metrics», in *GPU Computing Gems Jade Edition*, W.-mei W. Hwu, A c. di Boston: Morgan Kaufmann, 2012, pagg. 15–28.

[12] M. Lambertini, M. Magnani, M. Marzolla, D. Montesi, e C. Paolino, «Large-Scale Social Network Analysis», in *Large-Scale Data Analytics*, A. Gkoulalas-Divanis e A. Labbi, A c. di New York, NY: Springer New York, 2014, pagg. 155–187.

[13] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, e D. Chavarria-Miranda, «A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets», in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, 2009, pagg. 1–8, doi: 10.1109/IPDPS.2009.5161100.

[14] M. Magnani e M. Marzolla, «Path-Based and Whole-Network Measures», in *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj e J. Rokne, A c. di New York, NY: Springer New York, 2017, pagg. 1–16.

[15] A. McLaughlin e D. A. Bader, «Scalable and High Performance Betweenness Centrality on the GPU», in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pagg. 572–583, doi: 10.1109/SC.2014.52.

[16] NVIDIA, P. Vingelmann, e F. H. P. Fitzek, «CUDA, release: 11.2.142». https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf , 2020.

[17] G. Sabidussi, «The centrality index of a graph», *Psychometrika*, vol. 31, n. 4, pagg. 581–603, dic. 1966, doi: 10/fqhhrk.

[18] J. Siek, L.-Q. Lee, e A. Lumsdaine, *The boost graph library: user guide and reference manual*. Boston: Addison-Wesley, 2002.