UNIVERSITÁ DEGLI STUDI DI UDINE

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Progetto

per il corso di

Programmazione su Architetture Parallele

A.A. 2023-2024

Implementazione Parallela dell'algoritmo Push-Relabel per il problema Max-Flow/Min-Cut

Claudio Lanzaro Daniele Milo

26 Settembre 2024

Indice

Capitolo 1

Introduzione

1.1 Descrizione del progetto

TODO

1.2 Gruppo

Il progetto è stato svolto da un gruppo di due studenti composto da:

Matricola	Cognome e Nome	Indirizzo Mail
142252	Lanzaro Claudio	lanzaro.claudio@spes.uniud.it
142387	Milo Daniele	milo.daniele@spes.uniud.it

1.3 Consegna

Il lavoro svolto viene consegnato in un archivio **.zip** contenente:

- ...

Capitolo 2

Descrizione del problema

2.1 Minimum Cut Set

Un *taglio* S di un grafo non orientato $G = (V, E)$ è un sottoinsieme proprio di V ($S \in V$ e $S \neq \emptyset, V$). La dimensione del taglio S è il numero di archi tra S e il resto del grafo $V \setminus S$. Se il grafo è pesato, allora la dimensione del taglio corrisponde alla somma dei pesi degli archi che collegano le due parti.

Il taglio S è detto *minimo* se, tra tutti i tagli possibili all'interno del grafo G, è quello di dimensione minima.

Trovare il taglio minimo di un grafo non orientato e pesato sugli archi è un problema algoritmico fondamentale. Le sue applicazioni sono varie in campi come il routing nelle reti di telecomunicazione, la progettazione di circuiti, l'ottimizzazione delle risorse e la segmentazione di immagini.

2.1.1 Definizione formale

Il problema del taglio minimo (anche detto *Min Cut Problem*) è definito come segue:

Input: un grafo non orientato $G = (V, E)$. Due vertici $s, t \in V$. Pesi $w_{ij} \geq 0$ per ogni arco $ij \in E$.

Goal: trovare $S \in V$ tale che $s \in S, t \notin S$ e

$$\sum_{\substack{ij \in E: \\ i \in S \wedge j \notin S}} w(i, j) \tag{2.1}$$

ha valore minimo. Un caso speciale si verifica quando $w_{ij} = 1$ per tutti gli archi $ij \in E$ e, quindi, siamo interessati a trovare un taglio di cardinalità minima [3].

2.2 Maximum Flow Problem

Il *Minimum Cut* si connette al problema del *Maximum Flow* attraverso il teorema del max-flow/min-cut, il quale afferma che il valore del flusso massimo da s a t è uguale al peso del taglio minimo che separa s da t nel grafo. Quindi, risolvere il problema del *Minimum Cut* può essere ricondotto a calcolare il flusso massimo attraverso il grafo.

Il problema del massimo flusso è formalizzato come segue:

Input: una rete di flusso $G = (V, A, k)$ con k capacità degli archi. Due vertici $s, t \in V$.

Goal: trovare un flusso s-t di valore massimo.

2.3 Algoritmi

Per risolvere il problema del *Minimum Cut* e del *Maximum Flow*, sono stati sviluppati diversi algoritmi che sfruttano approcci diversi per migliorare l'efficienza e l'accuratezza nella risoluzione.

Gli algoritmi per il *Maximum Flow* sono tra i più studiati in informatica teorica e applicata, poiché risolvere questo problema consente di ottenere informazioni utili anche per il *Minimum Cut*, grazie al teorema del max-flow/min-cut. Alcuni dei principali algoritmi sono:

- **Algoritmo di Ford-Fulkerson**: è uno degli approcci più classici per risolvere il problema del massimo flusso. L'algoritmo costruisce incrementi di flusso lungo percorsi migliorativi fino a quando non è più possibile trovare percorsi che aumentino il flusso totale. L'implementazione di base ha complessità $O(|E| \cdot f)$, dove f è il valore del massimo flusso. Tuttavia, con l'uso della versione basata sull'algoritmo di Edmonds-Karp, che utilizza la ricerca in ampiezza (BFS) per trovare i percorsi migliorativi, la complessità diventa $O(|V| \cdot |E|^2)$.
- **Algoritmo di Push-Relabel**: invece di basarsi su percorsi migliorativi come Ford-Fulkerson, questo algoritmo si concentra su un nodo alla volta. La complessità è $O(|V|^2 \cdot |E|)$.

Grazie al teorema del max-flow/min-cut, risolvere il problema del *Maximum Flow* consente anche di risolvere il problema del *Minimum Cut*. Gli algoritmi di flusso massimo possono essere modificati per trovare direttamente il taglio minimo associato al flusso massimo. In particolare, al termine dell'algoritmo del massimo flusso, è possibile identificare il taglio minimo osservando i vertici raggiungibili dalla sorgente (o dal pozzo) nel grafo residuo. Per maggiori dettagli riguardo le reti di flusso si veda l'Appendice A.

2.4 Algoritmo Push-Relabel

Per questo progetto abbiamo deciso di trattare l'algoritmo Push-Relabel proposto da Goldberg e Tarjan [1], in quanto non solo ha la migliore complessità teorica e pratica, ma è anche particolarmente adatto alla parallelizzazione grazie alla sua struttura intrinseca e alla natura locale delle sue operazioni, risultando relativamente più facile da parallelizzare rispetto ad altri algoritmi basati su cammini aumentanti [2, 5].

L'algoritmo, conosciuto anche come preflow-push algorithm, si concentra su operazioni locali per aggiustare il flusso [2].

L'idea di base dell'algoritmo push-relabel è generare il maggior flusso possibile alla sorgente e gradualmente spingerlo verso il pozzo. Introduce il concetto di *excess flow* (flusso in eccesso), che consente a un vertice di ricevere più flussi in ingresso; in altre parole, consente a un vertice di avere più flusso in entrata di quanto ne esca durante la procedura push-relabel, e tale vertice viene chiamato *attivo*. Denotiamo con $e(v)$ il valore del flusso in eccesso sul vertice v. La procedura dell'algoritmo push-relabel consiste nel trovare i vertici attivi e applicare su di essi le operazioni di *push* e *relabel* fino a quando non rimangono più vertici attivi. Per trovare i vertici attivi iniziali, l'algoritmo spinge il maggiori quantitativo di flusso possibile dalla sorgente a tutti i vertici adiacenti, operazione chiamata *preflow* (preflusso). Un'operazione di push poi forza un vertice attivo a scaricare il suo flusso in eccesso e lo spinge verso i suoi vicini in G_f (grafo residuo). Per evitare infiniti push, l'algoritmo introduce anche la funzione altezza h su ciascun vertice. Inizialmente, l'altezza della sorgente è $|V|$, mentre l'altezza dei vertici rimanenti è 0. L'algoritmo, poi, forza un vertice attivo u a spingere il flusso verso il vertice v solo se $h(u) = h(v) + 1$. Se nessun vertice vicino soddisfa il vincolo, il vertice attivo aggiornerà la sua altezza trovando l'altezza minima h' dei suoi vicini e aggiornando $h(u) \leftarrow h' + 1$. Con l'operazione di relabel, l'altezza di un vertice attivo, che non può spingere il suo flusso in eccesso, aumenterà. Il vertice attivo verrà disattivato una volta che la sua altezza supererà $|V|$. L'intera procedura termina quando non ci sono più vertici attivi [2].

Algoritmo 1 Funzione *Init* per l'algoritmo push-relabel

for each $(x, y) \in E$ **do**
    $f(x, y) \leftarrow 1$
    $f(y, x) \leftarrow 0$
end for
for each $x \in V \setminus \{s\}$ **do**
    $e(x) \leftarrow 0$
    $h(x) \leftarrow 1$
end for
$e(s) \leftarrow \infty$
$h(s) \leftarrow |V|$

Algoritmo 2 Funzione *discharge(x)*

if $(\exists\ (x, y) \in E_f : h(x) = h(y) + 1$ **then**
    $push(x, y)$
else
    $relabel(x)$
end if

Algoritmo 3 Funzione *push(x, y)*

$\delta \leftarrow min\{u_f(x, y),\ e(x)\}$
$e(x) \leftarrow e(x) - \delta$
$e(y) \leftarrow e(x) + \delta$
$f(x, y) \leftarrow f(x, y) - \delta$
$f(y, x) \leftarrow f(y, x) + \delta$

Algoritmo 4 Funzione *relabel(x)*

$h(x) \leftarrow min\{h[y] : (x, y) \in E_f\} + 1$

Algoritmo 5 Algoritmo *push-relabel*

Init()
make set S empty
$S \leftarrow s$
while (S is not empty) **do**
    $x \leftarrow S$.pop()
    discharge(x)
    if (x is active node) **then**
        S.push(x)
    end if
end while

Algoritmo 6 Algoritmo *findMinCutSet(G_f,s)*

for each $v \in V$ **do**
    $visited[v] \leftarrow false$
end for
$visited[s] \leftarrow true$
make queue Q with s
while $Q \neq \emptyset$ **do**
    $u \leftarrow Q$.front()
    Q.pop()
    for each v adjacent to u **do** // $(v, u) \in E_f$
        if not $visited[v]$ **then**
            $visited[v] \leftarrow true$
            Q.push(v)
        end if
    end for
end while

Capitolo 3

Implementazione e ottimizzazioni

In questo capitolo presentiamo il percorso seguito per lo sviluppo di diverse versioni di un algoritmo parallelo per il calcolo del taglio minimo in un grafo. Il punto di partenza è stato l'implementazione di un algoritmo seriale, utilizzato come baseline per il confronto delle performance. Successivamente, si è passati a una prima versione parallela basata su CUDA, con l'obiettivo di sfruttare il calcolo parallelo per migliorare l'efficienza.

Dopo questa prima versione, è stato introdotto un miglioramento significativo attraverso l'uso della rappresentazione BCSR (Bidirectional Compressed Sparse Row), ottimizzata per ridurre il consumo di memoria e migliorare l'accesso ai dati. A seguire, ulteriori ottimizzazioni sono state implementate tramite un approccio work-balanced e vertex-centric, che hanno risolto il problema della scarsa efficienza dei thread inattivi, migliorando il bilanciamento del carico di lavoro tra i thread attivi. Questo approccio è stato proposto da un recente studio [2] e si è dimostrato efficace nel massimizzare l'utilizzo delle risorse hardware.

Infine, vengono presentati i dettagli dei test eseguiti utilizzando grafi generati casualmente e dataset standard in formato DIMACS. I test hanno valutato le prestazioni delle implementazioni in funzione del numero di nodi e della densità dei grafi, con un'analisi dettagliata dei tempi di esecuzione.

3.1 Versione seriale

La prima versione dell'algoritmo push-relabel implementata è stata quella seriale. Il codice sviluppato segue passo passo lo pseudocodice riportato nel Capitolo 2. Le performance misurate sono apparse sin da subito scarse con tempi che crescevano esponenzialmente all'aumentare del numero di nodi del grafo in input.

Le cause di questi risultati sono riconducibili in buona parte all'elevato numero di operazioni di relabel, molte delle quali inutili. Per ovviare al problema, Goldberg e Tarjan hanno proposto un'euristica chiamata *global relabeling* [1] che impedisce all'altezza dei nodi di aumentare velocemente e assegna ai nodi la minor altezza possibile.

La tecnica di global relabeling prevede l'esecuzione di una Breadth First Search (BFS) all'indietro (partendo dal nodo sink) nel grafo residuo, assegnando ai nodi nuove altezze corrispondenti al livello di ciascun nodo nell'albero BFS. La BFS richiede un tempo lineare pari a $O(m + n)$. In genere, il global relabeling viene applicato una volta ogni n operazioni di relabel. Questa euristica migliora notevolmente le prestazioni dell'algoritmo push-relabel.

Questa euristica, riportata nell'Algoritmo 7, è stata implementata in tutte le versioni parallele che verranno descritte in seguito in questo capitolo.

3.2 Prima versione parallela

Nella versione parallelizzata dell'algoritmo push-relabel si assume che il numero di thread in esecuzione sia |V| e che ciascuno di essi gestisca esattamente un nodo del grafo, comprese tutte le operazioni di push e relabel su di esso. In alcuni casi, più nodi possono essere gestiti da un solo thread.

Algoritmo 7 Euristica *Global Relabeling*

```
make queue Q empty
for each v ∈ V do
    scanned[v] ← false
end for
Q.push(t)
scanned[t] ← true
while Q ≠ ∅ do
    x ← Q.pop()
    current ← h(x)
    current ← current + 1
    for each (y, x) ∈ E_f : y not scanned do
        h(y) ← current
        scanned[y] ← true
        Q.pop()
    end for
end while
```

Sia x il thread in esecuzione che rappresenta il nodo $x \in V$. Ogni thread ha i seguenti attributi privati:

- la variabile e' che memorizza l'eccesso del nodo x;
- la variabile h'' che memorizza l'altezza del vicino y di x tale che $(x, y) \in E_f$;
- la variabile h' che memorizza l'altezza del vicino y' di x con altezza minore.

Altre variabili sono condivise tra tutti i thread in esecuzione. Tra queste ci sono gli array che contengono gli eccessi (e) e le altezze dei nodi (h), le capacità residue degli archi (c_f) e una variabile $TotalExcess$ equivalente al valore del flusso uscente dal nodo sorgente.

Tutti i thread in esecuzione hanno accesso alle variabili nella memoria globale, ma i loro compiti vengono eseguiti sequenzialmente grazie all'accesso atomico ai dati. L'ordine delle operazioni in questa sequenza non può essere previsto. Nonostante ciò, l'algoritmo può essere dimostrato corretto.

Inizialmente, l'operazione di inizializzazione ($Init$) viene eseguita dall'host (Algoritmo 8). Questo codice di inizializzazione è lo stesso della versione seriale dell'algoritmo push-relabel.

Il corpo principale dell'algoritmo è controllato dall'host. Il thread host esegue un ciclo `while` fino a quando il valore cumulativo degli eccessi memorizzati nella sorgente e nel pozzo raggiunge il valore di $TotalExcess$ (Algoritmo 11). A questo punto, tutto il flusso valido arriva al pozzo, mentre il resto del flusso torna alla sorgente. Di conseguenza, l'eccesso nel pozzo è uguale al valore del flusso massimo.

Nel primo passo del ciclo `while`, il thread host copia le altezze dei nodi nel device e avvia il kernel push-relabel. Quando il controllo ritorna al thread host, il "flusso" calcolato e le altezze dei nodi vengono copiati nella memoria della CPU e viene eseguita l'operazione di global-relabel.

Nel kernel push-relabel (Algoritmo 9), per $CYCLE$ iterazioni (dove $CYCLE$ è una costante che abbiamo definito pari a $|V|$), il thread cerca il nodo vicino con altezza minore verso cui effettuare un'operazione di push o di relabel. Il ciclo `while` nel kernel, quindi, si interrompe dopo un numero di iterazioni definito da $CYCLE$ e non quando il suo nodo diventa inattivo.

Al momento dell'esecuzione del global-relabeling (Algoritmo 10), poiché il ciclo `while` del kernel può terminare in qualsiasi momento, può verificarsi che la proprietà di alcuni archi residui $(x, y) \in E_f$ venga violata, cioè $h(x) > h(y) + 1$. Pertanto, prima di calcolare le nuove altezze dei nodi con il global-relabeling, tutti gli archi che violano la proprietà vengono corretti tramite il push del flusso. Gli eccessi dei nodi che non sono raggiungibili dalla BFS all'indietro devono essere sottratti da $TotalExcess$ poiché rappresentano un eccesso accumulato che non raggiungerà mai il pozzo.

3.2.1 Implementazione kernel Push-Relabel

Nota: tutti i nomi di funzioni che seguiranno fanno riferimento al codice consegnato.

Algoritmo 8 *Init()*

$h(s) \leftarrow |V|$
$e(s) \leftarrow 0$
for each $(u, y) \in E$ **do**
    $c_f(x, y) \leftarrow c_{xy}$
    $c_f(y, x) \leftarrow c_{yx}$
end for
for each $(s, x) \in E$ **do**
    $c_f(s, x) \leftarrow 0$
    $c_f(x, s) \leftarrow c_{xs} + c_{sx}$
    $e(s) \leftarrow c_{sx}$
    $TotalExcess \leftarrow TotalExcess + c_{sx}$
end for

Algoritmo 9 *PushRelabelKernel()*

$cycle =$ CYCLE
while $cycle > 0$ **do**
    if $e(x) > 0$ and $h(x) < |V|$ **then**
        $e' \leftarrow e(x)$
        $h' \leftarrow \infty$
        for each $(x, y) \in E_f$ **do**
            $h'' \leftarrow h(y)$
            if $h'' < h'$ **then**
                $y' \leftarrow y$
                $h' \leftarrow h''$
            end if
        end for
        if $h(x) > h'$ **then** *// Thread performs PUSH*
            $\delta \leftarrow min\{e', c_f(x, y)\}$
            AtomicAdd($c_f(y', x), \delta$)
            AtomicSub($c_f(x, y'), \delta$)
            AtomicAdd($e(y'), \delta$)
            AtomicSub($e(x), \delta$)
        else *// Thread performs RELABEL*
            $h(u) \leftarrow h' + 1$
        end if
    end if
    $cycle = cycle - 1$
end while

Algoritmo 10 *GlobalRelabel()*

for each $(x, y) \in E$ **do**
    if $h(x) > h(y) + 1$ **then**
        $e(x) \leftarrow e(x) - c_f(x, y)$
        $e(y) \leftarrow e(y) + c_f(x, y)$
        $c_f(y, x) \leftarrow c_f(y, x) + c_f(x, y)$
        $c_f(x, y) \leftarrow 0$
    end if
end for
do a backwards BFS from sink and assign the height function with each node's BFS tree level
if not all the nodes are scanned **then**
    for each $x \in V$ **do**
        if x is not scanned and not marked **then**
            mark x
            $TotalExcess \leftarrow TotalExcess - e(x)$
        end if
    end for
end if

Algoritmo 11 *Main()*

Init()
Copy e and c_f from the CPU's main memory to the CUDA device global memory
while $e(s) + e(t) < TotalExcess$ **do**
    Copy h from the CPU's main memory to the CUDA device global memory
    PushRelabelKernel()
    Copy e, c_f and h from the CUDA device global memory to the CPU's main memory
    GlobalRelabel()
end while

La funzione host `pushRelabel` gestisce l'intero processo dell'algoritmo, eseguendo i seguenti passaggi:

- preflow iniziale dai nodi sorgente;
- trasferimento delle variabili necessarie sulla memoria device;
- lancio del kernel `pushRelabelKernel` per eseguire il ciclo di push-relabel in parallelo;
- trasferimento delle variabili necessarie sulla memoria host;
- applicazione dell'operazione di global relabel per aggiornare le altezze dei nodi.

Gestione della memoria

Le variabili principali, come le matrici di capacità (`d_capacities`), flusso residuo (`d_residual`), altezza dei nodi (`d_height`) e flusso in eccesso (`d_excess`), vengono allocate nella memoria globale della GPU. I dati vengono trasferiti dalla memoria host a quella device tramite `cudaMemcpy` prima dell'esecuzione del kernel e successivamente copiati di nuovo alla memoria host per ulteriori elaborazioni.

Kernel

Il kernel viene eseguito tramite la funzione `cudaLaunchCooperativeKernel`, che sfrutta i cooperative groups per migliorare l'efficienza del calcolo parallelo. Il numero di blocchi (`num_blocks`) è determinato dal numero di streaming multiprocessors della GPU, con ciascun blocco composto da 256 thread (`block_size`), per ottimizzare il parallelismo.

Il kernel `pushRelabelKernel` implementa in modo parallelo l'algoritmo push-relabel, gestendo i nodi che hanno un eccesso di flusso. Ogni thread della GPU è responsabile di uno o più nodi, permettendo all'algoritmo di scalare su grafi di grandi dimensioni (tecnica *Grid-Stride loops*).

L'uso di cooperative groups consente ai thread di sincronizzarsi durante l'esecuzione del kernel. La funzione `grid.sync()` garantisce che tutte le operazioni di push e relabel siano completate prima di iniziare il ciclo successivo, mantenendo la coerenza dei dati condivisi tra i thread. Il kernel si concentra sui nodi attivi (esclusi quelli sorgente e destinazione), eseguendo le operazioni di push o relabel.

Le operazioni di push vengono eseguite tramite operazioni atomiche, utilizzando `atomicAdd` e `atomicSub` per aggiornare in sicurezza le matrici del flusso residuo e dell'eccesso. La sincronizzazione con `grid.sync()` dopo ogni ciclo assicura che tutti i thread completino le loro operazioni prima di passare al ciclo successivo. Questo è fondamentale per mantenere la coerenza dei dati condivisi tra i thread.

Sincronizzazione e gestione errori

Dopo il lancio del kernel, `cudaDeviceSynchronize` viene chiamato per assicurare il completamento di tutte le operazioni prima di proseguire. Eventuali errori durante l'esecuzione vengono rilevati con `cudaGetLastError`, fornendo messaggi diagnostici in caso di fallimento.

3.2.2 Test e miglioramenti

Questa prima versione parallela è stata testata su un insieme di grafi generati casualmente. Durante la fase iniziale di testing, grazie all'uso del profiler *nvprof*, abbiamo osservato che la maggior parte del tempo di esecuzione veniva impiegata nei trasferimenti di dati tra host e

device. Per ridurre tali tempi, abbiamo sperimentato diverse ottimizzazioni, tra cui l'uso della memoria page-locked e l'integrazione della memoria page-locked con quella mapped. Tuttavia, entrambe le versioni non hanno prodotto i miglioramenti attesi, anzi, i tempi di esecuzione sono persino aumentati. Pertanto, siamo tornati alla versione iniziale.

Inoltre, i test hanno evidenziato una limitazione significativa: i grafi utilizzabili come input non potevano avere dimensioni elevate, poiché la loro rappresentazione matriciale richiede uno spazio di memoria pari a $O(V^2)$.

Per superare questo vincolo, le prossime implementazioni adotteranno una rappresentazione dei dati più compatta.

3.3 Seconda versione parallela

Le precedenti versioni dell'algoritmo utilizzavano una matrice di adiacenza per rappresentare il grafo in input e quello residuo. Usare questo tipo di rappresentazione è comodo e richiede meno tempo in alcune fase del processo come la ricerca del nodo vicino con altezza minima. Tuttavia, come constatato nei primi test condotti, la complessità in termini di spazio è $O(V^2)$ e in presenza di grafi sparsi, o non particolarmente densi, provoca uno spreco di memoria impedendo la scalabilità delle soluzioni implementate.

Per ovviare al problema, nelle versioni seguenti abbiamo adottato una rappresentazione chiamata *Bidirectional Compressed Sparse Representation* (BCSR).

3.3.1 Bidirectional Compressed Sparse Representation

La *Bidirectional Compressed Sparse Representation* proposta da Hsieh et al. [2] è una evoluzione della classica rappresentazione CSR (Compressed Sparse Row).

Il formato CSR offre già un notevole risparmio di memoria quando i grafi da rappresentare sono sparsi ma ha un piccolo difetto: dato un nodo u, dopo aver trovare il vicino v con altezza minima, l'algoritmo push-relabel richiede di trovare l'arco (v, u) e questa operazione non è ottimizzata al meglio.

Con la BCSR si vuole migliorare questo aspetto. In particolare, i vicini (sia entranti che uscenti) vengono raggruppati e la lista *column* è ordinata in modo crescente per ID del nodo. Così facendo, si può utilizzare una binary search per la ricerca dei nodi e ottenere performance migliori.

In Figura 3.1 è riportato un esempio di grafo residuo rappresentato sia tramite matrice di adiacenza che tramite BCSR.

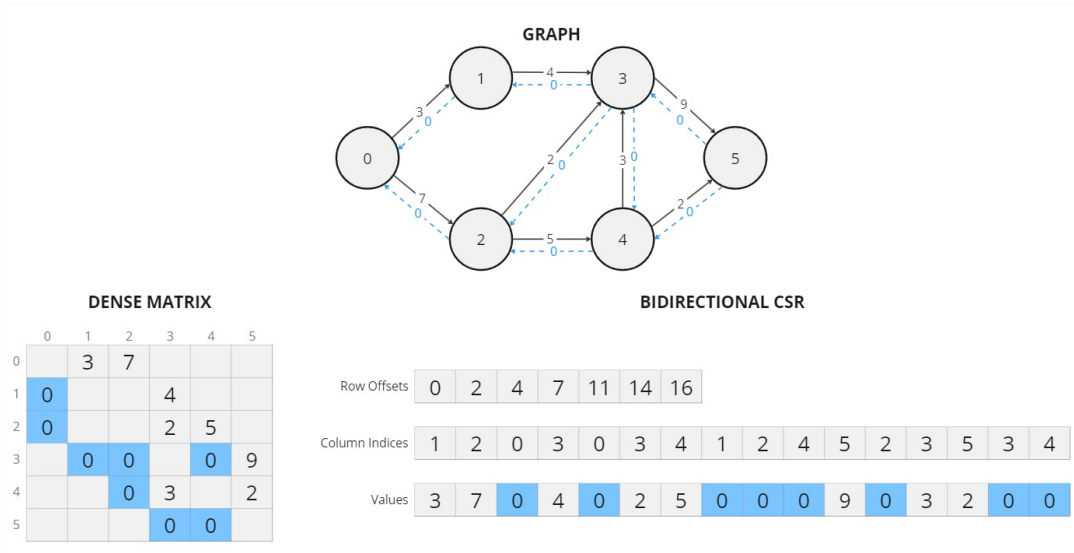

Figura 3.1: Matrice di adiacenza vs BCSR

3.3.2 Modifiche al codice

Con l'introduzione della nuova modalità di rappresentazione dei grafi, per adattare il codice della prima versione parallela, è stato sufficiente apportare delle piccole modifiche alle porzioni di codice che eseguivano accessi ai dati del grafo. Ad esempio, nei Listati 1 e 2 sono confrontati gli accessi ai dati nella prima versione e quello nella seconda.

```
// Ricerca nodo adiacente con altezza minore
for(v = 0; v < V; v++){
    if(d_residual[u*V + v] > 0){
        h2 = d_height[v];
        ...
    }
}
```

Listato 1: Ricerca nodo vicino con matrice d'adiacenza

```
// Ricerca nodo adiacente con altezza minore
for(int i = d_offset[u]; i < d_offset[u+1]; i++){
    v = d_column[i];
    if(d_residual[i] > 0){
        h2 = d_height[v];
        ...
    }
}
```

Listato 2: Ricerca nodo vicino con BCSR

Inoltre, dato che con BCSR trovare l'arco all'indietro non è immediato come nel caso con matrice di adiacenza, è stato necessario implementare un processo di ricerca apposito. Una prima soluzione è quella mostrata nel Listato 3: per trovare l'arco all'indietro $(minV, u)$ occorre controllare per ogni vicino di $minV$ se quest'ultimo è u, in questo caso si salva l'indice j dell'arco ricercato.

```
int backwardIdx = -1;

// Ricerca arco di ritorno
for(int j = d_offset[minV]; j < d_offset[minV+1]; j++){
    if(d_column[j] == u){
        backwardIdx = j;
        break;
    }
}
```

Listato 3: Ricerca lineare

Successivamente, per sfruttare tutti i vantaggi che la BCSR offre, abbiamo sostituito questa porzione di codice con una equivalente che implementa la ricerca binaria (Listato 4).

```
int backwardIdx = -1;

// Ricerca arco di ritorno usando la ricerca binaria
int left = d_offset[minV];
int right = d_offset[minV + 1] - 1;

while (left <= right) {
    int mid = left + (right - left) / 2;
    if (d_column[mid] == u) {
        backwardIdx = mid;  // Arco di ritorno trovato
        break;
    } else if (d_column[mid] < u) {
        left = mid + 1;     // Cerca nella parte destra
    } else {
        right = mid - 1;    // Cerca nella parte sinistra
    }
}
```

Listato 4: Ricerca binaria

3.4 Terza versione parallela

Come ultima versione parallela, abbiamo voluto implementare la soluzione proposta da Hsieh et al. in [2]. Si tratta di un approccio che combina l'utilizzo della rappresentazione BCSR e una distribuzione più equa del carico di lavoro tra i thread.

Nelle versioni parallele analizzate finora è stato adottato un approccio *thread-centric* in cui ogni thread gestiva le operazioni relative a un singolo nodo, indipendentemente dal fatto che fosse attivo o inattivo. Questo ha portato a uno spreco di risorse, poiché alcuni thread erano assegnati a nodi inattivi, mentre altri risultavano sovraccaricati, dovendo elaborare nodi con molti vicini e quindi richiedendo più lavoro.

In questa versione si introduce, invece, un approccio *vertex-centric* il quale si concentra esclusivamente sui nodi attivi distribuendo le risorse computazionali tra di essi. Hsieh et al. hanno chiamato questa versione *workload-balanced push-relabel algorithm* [2].

3.4.1 Approccio vertex-centric

L'algoritmo inizia assegnando tutti i thread alla scansione dei vertici per individuare quelli attivi, che vengono poi aggiunti alla coda dei vertici attivi (AVQ). Questo approccio garantisce una distribuzione uniforme del carico di lavoro tra i thread durante la fase di identificazione dei vertici attivi. Inoltre, i vertici attivi vengono raggruppati all'interno dell'AVQ, consentendo di assegnare ad ogni nodo attivo un gruppo di thread (tile) per cercare il vicino con altezza minima. In questo modo, il tempo di ricerca viene ridotto.

Nell'Algoritmo 12 è riportato lo pseudocodice che schematizza quanto appena detto.

Durante ciascuna iterazione, i vertici attivi vengono aggiunti alla coda AVQ tramite l'operazione AtomicAdd(). A questo punto, grazie all'inserimento di una sincronizzazione (grid.sync()), l'assegnazione dei thread può essere riorganizzata per ottimizzare la ricerca del vicino con altezza minima di ogni nodo attivo. Quando non ci sono più vertici attivi nella coda AVQ, il ciclo `while` (Algoritmo 9) viene interrotto anticipatamente, evitando iterazioni non necessarie. Per migliorare l'efficienza, si usa un warp come tile che viene assegnato a ciascun vertice attivo per parallelizzare la ricerca del vicino minimo. La memorizzazione contigua dei vicini nella struttura BCSR permette di sfruttare la coalescenza della memoria, velocizzando il processo. Dopo aver identificato il vicino, il thread con $localIdx$ pari a 0 all'interno del warp esegue le operazioni di aggiornamento del grafo ("push" o "relabel").

Nell'Algoritmo 13 viene descritto il processo di ricerca del vicino con altezza minima impiegando le tile. Si inizia con l'inizializzazione delle variabili necessarie e il calcolo del numero di iterazioni richieste per elaborare tutti i vicini del nodo utilizzando i thread contenuti nella tile. In questa fase preliminare si inizializza anche la memoria condivisa (shared memory), dove a ciascun thread viene assegnato lo spazio per tre interi: altezza, ID e indice del nodo vicino.

Algoritmo 12 *PushRelabelKernel()*

```
make a queue avq empty
// Scanning active vertices
for each u ∈ V do
    if e(u) > 0 and h(u) < |V| then
        pos ← AtomicAdd(avq, 1)
        avq[pos] ← u
    end if
end for
grid.sync()
// Processing only active vertices
for each u ∈ avq do
    // Searching for min height neighbor of u
    for each v ∈ neighbor(u) do
        min = TiledSearchNeighbor()
    end for
    tile.sync()
    // Only one thread per tile computes push or relabel operations
    if localIdx == 0 then
        if h(v) < h(min) then
            Push()
        else
            Relabel()
        end if
    end if
end for
```

Successivamente, per ogni iterazione prevista, si esaminano tutti i vicini, memorizzando solo le informazioni di quelli che soddisfano determinate condizioni. Una volta completata la scansione, viene effettuata una riduzione per ottenere le informazioni del nodo cercato. In seguito, uno dei thread della tile salva il risultato parziale prima di procedere all'iterazione successiva. Al termine di tutte le iterazioni, nelle variabili che contenevano i risultati parziali si troverà il risultato finale, che verrà infine restituito al chiamante.

Un'ultima sostanziale differenza rispetto alle versioni precedenti è l'implementazione in CUDA anche della procedura di global-relabeling. La struttura del codice è molto simile a quella vista nelle precedenti versioni.

Algoritmo 13 *TiledSearchNeighbor(tile, pos)*

```
idx ← tile.thread_rank() // thread index within the tile
tidx ← threadIdx.x // thread index within the grid
u ← avq[pos] // node whose neighbors will be scanned
degree ← offset[u + 1] − offset[u] // number of neighbors
numIters ← ⌈degree/tileSize⌉
initialize variables minH and minV // to save intermediate and final result
initialize shared memory variables s_height[tidx], s_vid[tidx], s_vidx[tidx]
for i = 0; i < numIters; i++ do
    vPos ← offset[u] + i ∗ tileSize + idx
    v ← column[vPos]
    if flows[vPos] > 0 and v ≠ source then
        s_height[tidx] ← d_height[v]
        s_vid[tidx] ← v
        s_vidx[tidx] ← vPos
    end if
    tile.sync()
    // Parallel reduction over shared memory values
    for s = tile.size()/2; s > 0; s >>= 1 do
        if idx < s then
            if s_height[tidx + s] < s_height[tidx] then
                s_height[tidx] ← d_height[tidx + s]
                s_vid[tidx] ← s_vid[tidx + s]
                s_vidx[tidx] ← s_vidx[tidx + s]
            end if
        end if
        tile.sync()
    end for
    tile.sync()
    // Only one thread updates the result of the current iteration
    if idx == 0 then
        if minH > s_height[tidx] then
            minH = s_height[tidx]
            minV = s_vid[tidx]
        end if
    end if
    tile.sync()
    reset variables for next iteration
    tile.sync()
end for
tile.sync()
return minV
```

3.5 Parallelizzazione ricerca del taglio minimo

Come anticipato nella Sezione 2.3, al termine dell'esecuzione dell'algoritmo push-relabel, quello che si può ottenere immediatamente è il valore del massimo flusso. Per ottenere il minimum cutset, invece, occorre effettuare una visita in ampiezza (BFS) a partire da nodo pozzo (*sink*). Nello specifico, l'insieme di taglio minimo sarà composto da tutti i nodi v tali per cui esiste un cammino $v \rightarrow^* sink$.

Inizialmente, è stata implementata una versione seriale (`findMinCutSetFromSink()`) che veniva eseguita sul grafo residuo restituito dall'algoritmo push-relabel. Durante i test, però, abbiamo riscontrato che la ricerca del minimum cutset impiegava, nel caso peggiore provato, decine di minuti rispetto ai pochi secondi impiegato da push-relabel.

Vista tale differenza, abbiamo pensato di applicare delle ottimizzazioni tra cui la parallelizzazione tramite OpenMP. Di seguito descriveremo 3 versioni: quella seriale, quella parallela su matrice di adiacenza e quella parallela su rappresentazione BCSR. Ognuna di queste implementazioni è stata utilizzata assieme alle versioni opportune dell'algoritmo push-relabel.

3.5.1 Versione seriale

Questa versione è stata la prima implementata e quella che ha avuto le performance peggiori. Si tratta del punto di partenza da cui abbiamo sviluppato le versioni successive. Nel Listato 5 riportiamo il codice della funzione.

```
std::vector<int> findMinCutSetFromSink(int V, int sink, int *offset,
                                       int *column, int *forwardFlow){
    std::vector<int> minCutSet;
    std::queue<int> q;
    std::vector<bool> visited(V, false);

    // BFS per trovare il taglio minimo a partire dal nodo sink
    minCutSet.push_back(sink);
    q.push(sink);
    visited[sink] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // Scansione dei vicini di u che hanno flusso verso u
        for (int v = 0; v < V; v++) {
            for (int i = offset[v]; i < offset[v+1]; i++) {
                if(column[i] == u && forwardFlow[i] > 0 && !visited[v]) {
                    minCutSet.push_back(v);
                    q.push(v);
                    visited[v] = true;
                }
            }
        }
    }

    return minCutSet;
}
```

Listato 5: findMinCutSetFromSink - implementazione seriale

3.5.2 Versione parallela su matrice di adiacenza

A partire dalla versione appena vista, abbiamo:

- sostituito la coda q per memorizzare i nodi della frontiera con un vettore *vertexList*;
- distribuito i nodi della frontiera su più thread;

- introdotto delle strutture locali ai thread per ridurre gli accessi concorrenti;
- protetto la lettura dell'attributo *visited* del nodo per evitare letture errate.

Per la distribuzione delle iterazioni del ciclo `for` tra i thread, abbiamo provato diverse tecniche di schedulazione, quella migliore è risultata essere `schedule(dynamic)`.

La lettura di `visited(v)` è stata inserita all'interno di una sezione critica per evitare race condition. Senza questa protezione, poteva verificarsi che un thread leggesse il valore `false` per un nodo subito prima che un altro thread aggiornasse lo stesso valore a `true`. Questa situazione avrebbe causato l'inserimento di duplicati dello stesso nodo nell'insieme del minimum cutset finale. La regione critica garantisce che l'accesso a `visited(v)` avvenga in modo sicuro e sincronizzato tra i thread, prevenendo tali inconsistenze.

Nel Listato 6 è riportato il codice completo.

3.5.3 Versione parallela su rappresentazione BCSR

Per i dati rappresentati tramite il formato BCSR, l'approccio adottato è stato leggermente diverso. A causa della particolare struttura di questa rappresentazione, come già accennato, la ricerca dei nodi v, dato u, tali per cui esiste un arco (v, u) risulta più onerosa in termini computazionali rispetto al caso della matrice di adiacenza.

Per ovviare a questo problema, tra le soluzioni analizzate, la più efficace è risultata essere quella illustrata nel Listato 7. In questo caso, prima di iniziare la ricerca a partire dal *sink*, viene calcolato il grafo residuo trasposto. Questo permette di evitare il problema della ricerca degli archi entranti, poiché una volta trasposto il grafo, sarà sufficiente seguire gli archi uscenti da ogni nodo incontrato durante la visita.

A parte questa modifica, il resto della procedura segue la stessa logica della versione basata sulla matrice di adiacenza, con alcuni adattamenti necessari per gestire la lettura dei dati secondo la struttura del formato BCSR.

```cpp
std::vector<int> findMinCutSetFromSinkOMP(int n, int t, int *residual){
    std::vector<int> minCutSet;
    std::vector<int> vertexList;
    std::vector<bool> visited(n, false);

    minCutSet.push_back(t);
    vertexList.push_back(t);
    visited[t] = true;

    while (!vertexList.empty()) {
        std::vector<int> newVertexList;

        #pragma omp parallel
        {
            std::vector<int> localVertexList;
            std::vector<int> localMinCutSet;

            #pragma omp for nowait schedule(dynamic)
            for (int i = 0; i < vertexList.size(); i++) {
                int u = vertexList[i];

                for (int v = 0; v < n; v++) {
                    bool shouldAdd = false;

                    #pragma omp critical (checkVisited)
                    {
                        if (!visited[v] && residual[v*n + u] > 0) {
                            visited[v] = true;
                            shouldAdd = true;
                        }
                    }

                    if(shouldAdd){
                        localMinCutSet.push_back(v);
                        localVertexList.push_back(v);
                    }
                }
            }

            #pragma omp critical (insert)
            {
                minCutSet.insert(minCutSet.end(),
                    localMinCutSet.begin(), localMinCutSet.end());
                newVertexList.insert(newVertexList.end(),
                    localVertexList.begin(), localVertexList.end());
            }
        }
        vertexList = newVertexList;
    }

    return minCutSet;
}
```

Listato 6: findMinCutSetFromSinkOMP - implementazione parallela su matrice di adiacenza

```cpp
std::vector<int> findMinCutSetFromSinkOMP(int V, int E, int sink,
                                int *offset, int *column, int *forwardFlow){
    int *t_offset = (int*)malloc((V+1)*sizeof(int));
    int *t_column = (int*)malloc(E*sizeof(int));
    int *t_forwardFlow = (int*)malloc(E*sizeof(int));
    for(int i = 0; i < V+1; i++){
        t_offset[i] = 0;
    }
    for(int i = 0; i < E; i++){
        t_column[i]= 0;
        t_forwardFlow[i] = 0;
    }

    computeTranspose(V, E, offset, column, forwardFlow, t_offset,
                        t_column, t_forwardFlow);

    std::vector<int> minCutSet;
    std::vector<int> vertexList;
    bool *visited = (bool*)malloc(V*sizeof(bool));
    for(int i = 0; i < V; i++) {
        visited[i] = false;
    }
    minCutSet.push_back(sink);
    vertexList.push_back(sink);
    visited[sink] = true;
    while (!vertexList.empty()) {
        std::vector<int> newVertexList;
        #pragma omp parallel
        {
            std::vector<int> localVertexList;
            std::vector<int> localMinCutSet;
            #pragma omp for nowait schedule(dynamic)
            for (int i = 0; i < vertexList.size(); i++) {
                int u = vertexList[i];
                for (int j = t_offset[u]; j < t_offset[u+1]; j++) {
                    int v = t_column[j];
                    bool shouldAdd = false;
                    #pragma omp critical (checkVisited)
                    {
                        if(!visited[v] && t_forwardFlow[j] > 0) {
                            visited[v] = true;
                            shouldAdd = true;
                        }
                    }
                    if(shouldAdd) {
                        localMinCutSet.push_back(v);
                        localVertexList.push_back(v);
                    }
                }
            }
            #pragma omp critical (insert)
            {
                minCutSet.insert(minCutSet.end(),
                        localMinCutSet.begin(), localMinCutSet.end());
                newVertexList.insert(newVertexList.end(),
                        localVertexList.begin(), localVertexList.end());
            }
        }
        vertexList = newVertexList;
    }
    return minCutSet;
}
```

Listato 7: findMinCutSetFromSinkOMP - implementazione parallela su rappresentazione BCSR

3.6 Test e analisi delle performance

In ogni fase dello sviluppo delle varie versioni sono stati condotti dei test per verificare la correttezza del codice, individuare le parti da ottimizzare e confrontare le versioni finali per decretarne la migliore. In questa sezione descriveremo i dati utilizzati come input, le tecniche di misurazione dei tempi di esecuzione e gli strumenti per la visualizzazione dei dati ottenuti.

3.6.1 Dati di input

Durante lo svolgimento dei test sono stati impiegati 3 diversi gruppi di istanze di grafi:

- grafi generati casualmente con numero di nodi crescente e densità costante per misurare i tempi di esecuzione all'aumentare della dimensione dell'input;
- grafi generati casualmente con numero di nodi costante e densità crescente per confrontare le performance dei due tipi di rappresentazione dei grafi;
- grafi reperiti da un dataset online per testare gli algoritmi su istanza reali.

Generazione grafi casuali

Per la generazione dei grafi casuali abbiamo scritto un breve jupyter notebook in python e sfruttato alcune funzionalità della libreria *networkx*. In particolare, è stata implementata la funzione `generate_random_graph` (Listato 8), la quale utilizza il modello di grafi casuali Erdős-Rényi con una probabilità p di creare un arco tra ciascuna coppia di nodi. I parametri di input per la funzione includono:

- `n`, il numero di nodi nel grafo;
- `p`, la probabilità con cui ogni arco viene creato tra due nodi;
- `min_capacity` e `max_capacity`, i limiti inferiore e superiore per la capacità degli archi;
- `directed`, un parametro booleano che determina se il grafo sarà diretto o non diretto.

L'esecuzione della funzione genera un grafo con capacità casuali assegnate ad ogni arco. Un aspetto cruciale della generazione dei grafi è la connettività: per ogni istanza generata ci siamo assicurati che fosse connessa.

Per il primo gruppo di grafi (nodi crescenti e densità costante) come valore di densità abbiamo scelto $p = 2(log(n)/n)$ poiché, secondo la teoria, la probabilità che un Erdős-Rényi random graph sia connesso è molto alta quando $p > log(n)/n$.

Per il secondo gruppo, invece, i valori scelti sono stati $n = 1000$ e $p \in [0, 1]$ con passo 0.1.

```python
def generate_random_graph(n, p, min_capacity, max_capacity, directed=False):
    G = nx.gnp_random_graph(n, p, directed=directed, seed=7)

    for edge in G.edges:
        capacity = random.randint(min_capacity, max_capacity)
        G.edges[edge]['capacity'] = capacity

    return G
```

Listato 8: Funzione per generare un grafo casuale con il modello Erdős-Rényi

Una volta ottenuto il grafo generato, abbiamo scelto di salvarlo in un file di testo seguendo il formato:

- una riga per indicare il numero di nodi presenti (es. `n 500` indica 500 nodi con ID da 0 a 499);
- una riga per ogni arco e la relativa capacità (es. `e 1 2 5` indica arco da nodo 1 a nodo 2 con capacità 5).

Grafi da dataset online

I dati relativi al terzo gruppo di istanze è stato reperito al link `https://vision.cs.uwaterloo.ca/data/maxflow`. Si tratta di una pagina web in cui sono presenti molte istanze di grafi pesati raggruppati in dataset. Il dataset che abbiamo utilizzato per i test è chiamato *BVZ-tsukuba*.

I grafi sono salvati nel formato standard DIMACS, una sua descrizione è reperibile alla pagina `https://lpsolve.sourceforge.net/5.5/DIMACS_maxf.htm`.

Lettura grafi da file

Come si può evincere da quanto appena detto, essendo presenti due formati di file in input, è stato necessario implementare anche due funzioni C++ per il caricamento dei dati prima dell'esecuzione dell'algoritmo push-relabel.

La scelta di quale metodo chiamare è fatta in automatico sulla base dell'estensione del file passato al programma. Infatti, i grafi generati casualmente sono salvati in file `.txt` mentre i file in formato DIMACS hanno estensione `.max`.

3.6.2 Misurazione tempi

Per tutte le versioni dell'algoritmo implementate abbiamo misurato il tempo di inizializzazione (creazione e popolamento delle strutture dati e eventuali trasferimenti al device), tempo di esecuzione (dell'intero algoritmo push-relabel) e tempo totale (somma dei precedenti).

Versione seriale

Nella implementazione seriale, per la rilevazione dei tempi, abbiamo utilizzato le funzioni della libreria `chrono`. Nel Listato 9 è riportato uno schema semplificato di come la misurazione è avvenuta.

```
const auto start = chrono::high_resolution_clock::now();

[inizializzazione]

const auto endInitialization = chrono::high_resolution_clock::now();

[Esecuzione algoritmo push-relabel]

const auto end = chrono::high_resolution_clock::now();

double initializationTime = chrono::duration_cast<chrono::microseconds>
                                (endInitialization - start).count()/1000.0;
double executionTime = chrono::duration_cast<chrono::microseconds>
                                (end - endInitialization).count()/1000.0;
double totalTime = chrono::duration_cast<chrono::microseconds>
                                (end - start).count()/1000.0;


```

Listato 9: Schema misurazione tempi (versione seriale)

Versioni parallele

Nelle implementazioni parallele, la misurazione dei tempi è avvenuta tramite i `cudaEvent`. Nel Listato 10 è riportato uno schema esemplificativo.

```
//Dichiarazione degli eventi per la misurazione del tempo
cudaEvent_t startEvent, endInitializationEvent, endEvent;
cudaEventCreate(&startEvent);
cudaEventCreate(&endInitializationEvent);
cudaEventCreate(&endEvent);

...

// Primo evento per la misurazione del tempo
cudaEventRecord(startEvent, 0);

[inizializzazione]

// Secondo evento per la misurazione del tempo
cudaEventRecord(endInitializationEvent, 0);

[Esecuzione algoritmo push-relabel]

// Terzo evento per la misurazione del tempo
cudaEventRecord(endEvent, 0);

// Attendo la fine dell'evento endEvent
cudaEventSynchronize(endEvent);

// Misurazione del tempo
float initializationTime = 0.0f;
float executionTime = 0.0f;
float totalTime = 0.0f;
cudaEventElapsedTime(&initializationTime, startEvent, endInitializationEvent);
cudaEventElapsedTime(&executionTime, endInitializationEvent, endEvent);
cudaEventElapsedTime(&totalTime, startEvent, endEvent);

// Distruzione degli eventi
cudaEventDestroy(startEvent);
cudaEventDestroy(endInitializationEvent);
cudaEventDestroy(endEvent);
```

Listato 10: Schema misurazione tempi (versione parallela)

3.6.3 Esecuzione e analisi

Tutti i test eseguiti hanno prodotto dei risultati che sono stati salvati in opportuni file in formato JSON. Ognuno di essi conteneva:

- valore del massimo flusso;
- insieme dei nodi del minimum cutset;
- tempo di inizializzazione;
- tempo di esecuzione;
- tempo totale;
- numero di nodi del grafo;
- numero di archi.

Tutti questi file sono stati poi analizzati tramite un jupyter notebook in python per generare dei grafici che saranno mostrati nel Capitolo 4.

Per garantire risultati più affidabili ed eliminare possibili interferenze che avrebbero potuto influenzare i risultati, ciascun test è stato eseguito 10 volte e successivamente è stata calcolata la media dei tempi registrati.

Inoltre, tutte le sessioni di test sono avvenute sulla macchina *CUDA-srv* messa a disposizione dall'università con le seguenti caratteristiche:

- Intel Xeon W-3323 CPU @ 3.50GHz
- RAM 64GB
- Nvidia GeForce RTX 2080 Ti

Capitolo 4

Risultati

In questo capitolo verranno discussi i principali risultati riscontrati durante la fase di testing.

Per facilitare la lettura dei grafici, si ricorda che:

serial è la versione seriale, la prima implementata;

parallel è la prima versione parallela, i grafi sono rappresentati tramite matrice di adiacenza;

parallelbcsrtc è la seconda versione parallela, molto simile alla precedente, viene adattata per la rappresentazione BCSR includendo le ottimizzazioni che quest'ultima consente;

parallelbcsrvc è l'ultima versione implementata, la versione *work balanced*, e applica alla versione precedente diverse ottimizzazioni per sfruttare al massimo le risorse computazionali.

Nella prima parte metteremo a confronto le varie implementazioni su input in cui il numero di nodi è variabile e la densità è costante. Viceversa, nella seconda parte analizzeremo come la variazione di densità influisce sui tempi di esecuzione.

4.1 Risultati per quantità nodi variabile

Di seguito sono riportati due grafici che mettono a confronto le diverse implementazioni sia nel caso di grafi diretti (Figura 4.1) che in quello di grafi indiretti (Figura 4.2). I grafici rappresentano esclusivamente i tempi di esecuzione dell'algoritmo, sono esclusi quindi i tempi di inizializzazione e salvataggio dei risultati.

I valori di tutti gli assi sono espressi in scala logaritmica. Altrimenti, vista la grande differenza tra i valori, alcune linee risulterebbero appiattite e sovrapposte, rendendo difficile l'analisi dei risultati.

Sulle ordinate sono riportati i tempi espressi in millisecondi mentre sulle ascisse il numero di nodi.

Prima di passare alle osservazioni sui grafici, facciamo notare che non tutte le versioni sono state testate su tutte le istanze a disposizione per i seguenti motivi:

- la versione seriale è risultata essere talmente inefficiente (in termini di tempo) da consentirne l'esecuzione solamente con piccole istanze;
- la prima versione parallela, usando la matrice di adiacenza come metodo di rappresentazione dei dati, è risultata limitata nell'utilizzo della memoria al punto da renderne impossibile l'esecuzione su istanze con più di 30mila nodi.

Dunque, osservando i grafici si nota quanto segue:

- la versione seriale risulta essere competitiva solamente quando si hanno qualche decina di nodi. Dopodiché, qualsiasi versione parallela diventa preferibile;
- le versioni parallele, invece, tra di loro hanno tempi comparabili per grafi di piccole dimensioni, mentre per istanze di medie/grandi dimensioni si iniziano a notare delle differenze;
- in generale l'implementazione migliore sembrerebbe l'ultima proposta (approccio *work balanced*);

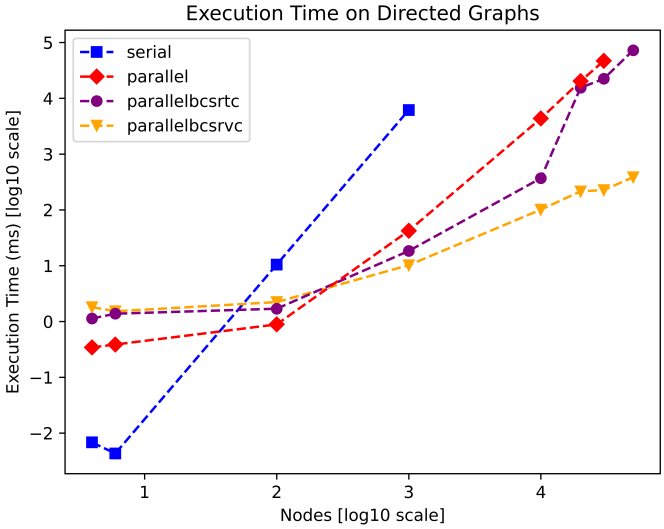

Figura 4.1: Tempi di esecuzione con grafi diretti

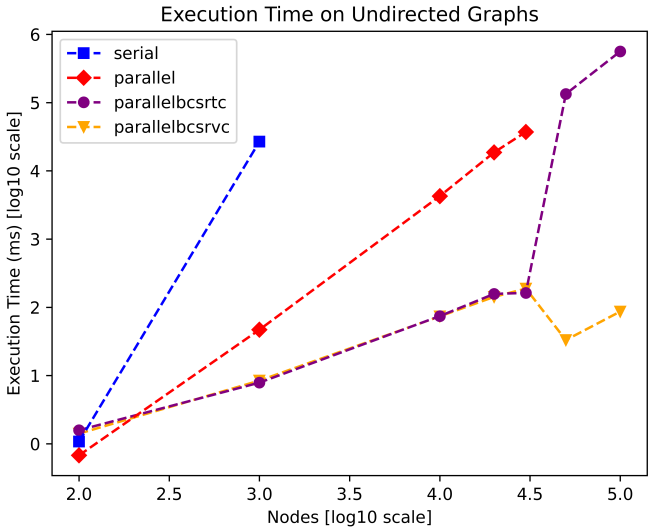

Figura 4.2: Tempi di esecuzione con grafi non diretti

- la versione parallela con BCSR (linea viola), all'aumentare delle dimensioni del grafo, sembrerebbe avvicinarsi ai tempi della versione parallela con matrice di adiacenza (linea rossa). Purtroppo non è possibile valutare con esattezza il loro comportamento, essendo la prima versione parallela fortemente limitata dal consumo di memoria.

4.2 Risultati per densità variabile

In questa sezione analizziamo come la densità del grafo influisce sui tempi di esecuzione dell'algoritmo. Il grafico in Figura 4.3 mostra i tempi di esecuzione delle diverse versioni dell'algoritmo su grafi diretti con densità variabile.

In questo caso, sull'asse X sono riportati i valori di densità, mentre sull'asse Y i tempi di esecuzione in millisecondi. Ricordiamo che i grafi usati in input avevano un numero di nodi fisso pari a 1000 e una densità variabile tra 0.1 e 1 (grafo completo).

Dal grafico si può notare che:

- la versione seriale non è presente poiché si è dimostrata essere inefficiente anche con grafi relativamente di piccole dimensioni e densità ridotta;

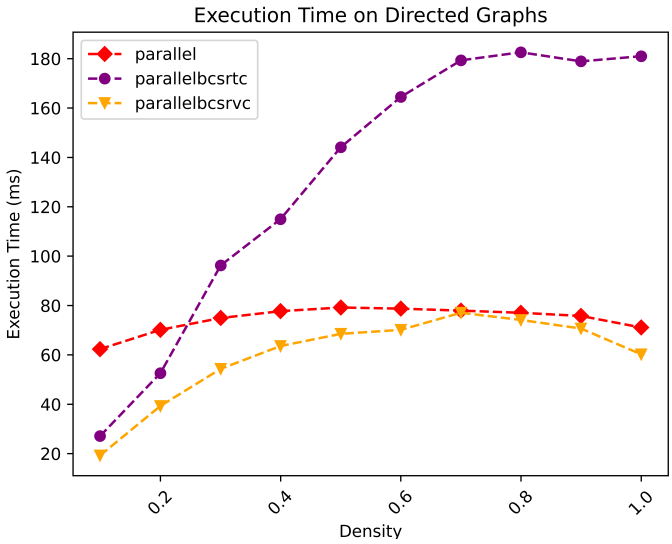

Figura 4.3: Tempi di esecuzione con grafi diretti a densità variabile

- la versione parallela con matrice di adiacenza (linea rossa) ha tempi di esecuzione pressoché costanti, indipendentemente dalla densità del grafo. Questo è dovuto al fatto che l'algoritmo che usa la matrice di adiacenza ha un costo computazionale che dipende dal numero di nodi e non dalla densità del grafo (la matrice ha sempre dimensione $|V| \times |V|$ indipendentemente dalla densità);
- la versione parallela con BCSR (linea viola) ha tempi di esecuzione crescenti all'aumentare della densità del grafo. Questo è dovuto al fatto che la rappresentazione BCSR è vantaggiosa per grafi sparsi e non densi. In particolare, all'aumentare della densità, lo stesso algoritmo con rappresentazione BCSR diventa meno efficiente rispetto a quello basato su matrice di adiacenza;
- la versione parallela *work balanced* (linea gialla) ha tempi di esecuzione crescenti ma mai superiori rispetto alla versione con matrice di adiacenza. Nello specifico, questa implementazione risulta essere vantaggiosa per grafi a bassa densità, mentre i tempi risultano essere comparabili per grafi a più alta densità. Questo è dovuto al fatto che l'implementazione *work balanced* combinata con la rappresentazione BCSR è ottimizzata per sfruttare al meglio le risorse computazionali.

4.3 Considerazioni

I test su grafi con densità e numero di nodi crescente offrono una valutazione dettagliata dell'algoritmo Push-Relabel parallelo, evidenziando vantaggi e limiti legati sia alla natura dei grafi sia alle strutture dati.

Con grafi a numero di nodi crescente, le versioni parallele superano presto quella seriale, con l'implementazione work-balanced che si distingue su istanze di medie e grandi dimensioni. Tuttavia, alcuni problemi come quelli legati alla memoria (ad esempio con la matrice di adiacenza) possono influire sulle prestazioni.

Nei grafi a densità crescente, la scelta delle strutture dati da utilizzare è cruciale. La versione con matrice di adiacenza ha tempi costanti, indipendenti dalla densità, mentre la versione BCSR diventa meno efficiente con grafi densi. L'implementazione work-balanced rimane vantaggiosa nei grafi a bassa densità, ma si allinea alla matrice di adiacenza con grafi più densi.

In sintesi, prestazioni e scalabilità dipendono dalla scelta delle strutture dati e dalla gestione del parallelismo.

Capitolo 5

Conclusioni

Il progetto ha avuto come obiettivo principale l'implementazione parallela dell'algoritmo Push-Relabel per la risoluzione del problema Max-Flow/Min-Cut. Attraverso un approccio incrementale, abbiamo sviluppato diverse versioni dell'algoritmo, migliorandone progressivamente l'efficienza e la scalabilità, a partire da una versione seriale fino ad arrivare a soluzioni parallele ottimizzate su GPU.

Sono state proposte tre diverse implementazioni parallele, ognuna con caratteristiche e prestazioni differenti. La prima implementazione, basata sulla matrice di adiacenza, è risultata essere la più semplice da realizzare, ma anche la meno efficiente in termini di memoria utilizzata. La seconda implementazione, strutturalmente simile alla precedente ma adattata sulla rappresentazione BCSR, ha permesso di ridurre il consumo di memoria rispetto alla prima implementazione, ma non ha portato a miglioramenti significativi in termini di tempo. Infine, la terza implementazione, basata sulla rappresentazione BCSR e ottimizzata per bilanciare il lavoro tra i vari thread della GPU, ha permesso di ottenere i migliori risultati in termini di tempo di esecuzione.

I test effettuati su diverse tipologie di grafi, sia reali che generati casualmente, hanno evidenziato significativi miglioramenti prestazionali nelle versioni parallele rispetto alla versione seriale. In particolare, l'adozione della rappresentazione Bidirectional Compressed Sparse Row (BCSR) e l'implementazione di un approccio workload-balanced hanno permesso di massimizzare l'efficienza nell'uso delle risorse computazionali.

Le analisi delle performance dimostrano che la versione parallela work-balanced si comporta in modo eccellente su grafi di grandi dimensioni e con bassa densità, pur mantenendo prestazioni competitive anche in scenari di alta densità. Tuttavia, la scelta della struttura dati più adatta, come la matrice di adiacenza o la BCSR, dipende strettamente dalle caratteristiche del grafo considerato.

In conclusione, il progetto ha dimostrato come sia possibile ottenere un'importante riduzione dei tempi di calcolo grazie alla parallelizzazione e a tecniche avanzate di rappresentazione dei dati. L'algoritmo Push-Relabel si conferma adatto a essere parallelizzato, rendendo possibili applicazioni efficienti su grafi di grandi dimensioni nel campo dell'ottimizzazione delle reti di flusso e altri ambiti correlati.

Appendice A

Reti di flusso

Formalmente una *rete di flusso* è definita come un grafo orientato $G = (V, E)$ tale che

- Ad ogni arco $(u, v) \in E$ e associato un valore non negativo, $c(u, v) \geq 0$, detto *capacità* dell'arco. Se il nodo (u, v) non è presente in E allora assumiamo che l'arco (u, v) abbia capacità nulla, cioè $c(u, v) = 0$.
- Nella rete di flusso ci sono due vertici speciali: il vertice *sorgente* (o *source*), indicato con il simbolo s, ed il vertice *pozzo* (o *sink*), indicato con il simbolo t.
- Ogni vertice giace su qualche cammino dalla sorgente al pozzo. Il grafo è quindi connesso e pertanto vale $|E| \geq |V| - 1$.

La Figura A.1 mostra un esempio di rete di flusso. La sorgente ed il pozzo sono rappresentati, rispettivamente, dai simboli s e t. Ad ogni arco $(u, v) \in E$ è associato il valore $c(u, v)$.

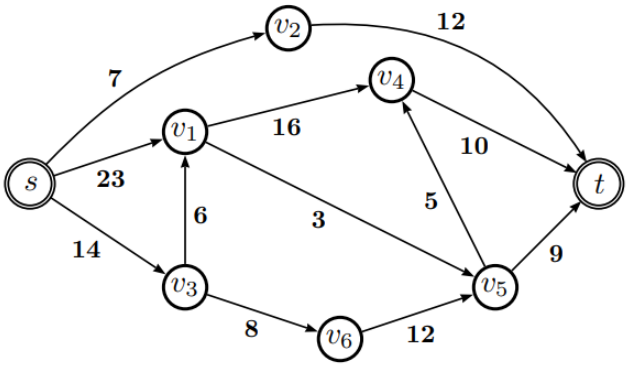

Figura A.1: Rete di flusso

A.1 Flusso

Per *flusso* in una rete si intende una funzione $x : E \to \mathbb{R}$ che rispetta i vincoli di capacità del grafo ovvero:

$$0 \leq x_{ij} \leq c(i, j) \quad \forall (i, j) \in E \tag{A.1}$$

Con il termine *divergenza* di un nodo i si intende la differenza tra il flusso uscente e quello entrante ovvero, dato un nodo $v \in V$ e un flusso x in G:

$$\Delta_x(v) = \sum_{\substack{j \in V \\ (v,j) \in E}} x_{vj} - \sum_{\substack{i \in V \\ (i,v) \in E}} x_{iv} \tag{A.2}$$

Si dice che il flusso x è conservato nel nodo v (o che il flusso x soddisfa i vincoli di conservazione del flusso) se $\Delta_x(v) = 0$. Un flusso che è conservato per ogni nodo è detto *circolazione*. Una *circolazione* può essere vista come uno scambio di beni in cui nulla viene mai creato o distrutto, ma è semplicemente passato da un nodo all'altro.

Se un flusso non è una circolazione, assumendo $S := \{v : \Delta_x(v) > 0\}$ e $T := \{v : \Delta_x(v) < 0\}$, possiamo interpretare il flusso come uno spostamento di beni in quantità $\phi = \sum_{v \in S} \Delta_x(v)$ dai nodi in S a quelli in T. Infatti, tutto il flusso uscente da S entrerà nei nodi in T.

Se consideriamo s (source) e t (sink) due nodi distinti in V, ogni flusso che è conservato in ogni nodo $v \neq s, t$ è chiamato *flusso s-t* e il suo valore è definito come:

$$\phi_x = \Delta_x(s) \tag{A.3}$$

Il problema del trovare il flusso s-t con valore massimo (dati s e t) all'interno di una rete è chiamato *problema del massimo flusso* [3].

A.2 Rete residua

Sia f un flusso in una rete (G, c), e si consideri una coppia di vertici $u, v \in V$. La quantità di flusso che possiamo inviare da u a v prima di superare la capacità $c(u, v)$ è la *capacità residua* di (u, v), ed è data da

$$c_f(u, v) = c(u, v) - f(u, v) \tag{A.4}$$

Data una rete di flusso (G, c) ed un flusso f, la rete residua di G indotta da f è $G_f = (V, E_f)$, dove

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} \tag{A.5}$$

Ogni arco della rete, chiamato *arco residuo*, può ammettere un flusso strettamente positivo. Un arco (u, v) può essere un arco residuo in E_f , anche se esso non fosse un arco in E, ma purché $(v, u) \in E$. Ciò implica che non sempre vale la relazione $E_f \subseteq E$. Però si ha sempre

$$|E_f| \leq 2 \cdot |E| \tag{A.6}$$

Si osservi che la rete residua G_f è essa stessa una rete di flusso avente capacità definita dalla funzione c_f.

Appendice B

Tecniche di Rappresentazione di Grafi

In questo capitolo sono descritte due delle principali tecniche utilizzate per rappresentare i grafi: la rappresentazione a matrice di adiacenza e la rappresentazione *Compressed Sparse Row*.

B.1 Rappresentazione dei Grafi con Matrici di Adiacenza

Una matrice di adiacenza è una rappresentazione comunemente utilizzata per descrivere un grafo $G = (V, E)$, dove V è l'insieme dei vertici e E è l'insieme degli archi (o collegamenti) tra i vertici. Questa tecnica rappresenta il grafo come una matrice quadrata di dimensioni $|V| \times |V|$, con le seguenti caratteristiche:

- ogni riga e colonna della matrice corrisponde a un vertice del grafo;
- il valore in posizione (i, j) della matrice indica se esiste un arco tra il vertice i e il vertice j;
- per grafi non pesati, l'elemento $A[i][j]$ è uguale a 1 se esiste un arco tra i e j, altrimenti è uguale a 0;
- per grafi pesati, l'elemento $A[i][j]$ contiene il peso dell'arco tra i e j (se esiste), mentre un valore predefinito, come ∞ o 0, è usato per indicare l'assenza di un arco;
- per grafi non orientati, la matrice di adiacenza è simmetrica rispetto alla diagonale principale, poiché un arco tra i e j implica anche un arco tra j e i.

Questo tipo di rappresentazione è particolarmente utile per grafi densi, dove la maggior parte dei vertici sono connessi tra loro, poiché richiede $O(|V|^2)$ spazio di memoria.

La matrice di adiacenza offre un modo semplice e diretto per eseguire operazioni come la verifica dell'esistenza di un arco tra due vertici, con una complessità $O(1)$. Tuttavia, può diventare inefficiente in termini di spazio di memoria per grafi molto sparsi, dove il numero di archi è significativamente inferiore a $|V|^2$. In questi casi, altre tecniche di rappresentazione, come le liste di adiacenza o il formato CSR per matrici sparse, possono essere più appropriate.

B.2 Compressed Sparse Row

Una matrice sparsa memorizzata in formato CSR (*Compressed Sparse Row*) è una rappresentazione compatta che si basa sul memorizzare solo gli elementi non nulli della matrice. Questa tecnica è particolarmente utile per matrici sparse di grandi dimensioni, dove la maggior parte degli elementi è zero, riducendo drasticamente lo spazio di memoria richiesto rispetto alla memorizzazione di una matrice completa. Nel formato CSR, la matrice è rappresentata dai seguenti parametri:

- il numero di righe nella matrice;
- il numero di colonne nella matrice;
- il numero di elementi non nulli (nnz) nella matrice;

- i puntatori all'array degli offset delle righe, di lunghezza pari al numero di righe +1, che rappresentano la posizione iniziale di ciascuna riga negli array delle colonne e dei valori;
- i puntatori all'array degli indici delle colonne, di lunghezza *nnz*, che contiene gli indici delle colonne degli elementi corrispondenti nell'array dei valori;
- i puntatori all'array dei valori, di lunghezza *nnz*, che contiene tutti i valori non nulli della matrice, ordinati in base alle righe.

La rappresentazione CSR permette di effettuare operazioni come il prodotto matrice-vettore in modo più efficiente, poiché si evita di processare esplicitamente gli elementi nulli. Inoltre, poiché gli elementi non nulli sono memorizzati in forma compatta, si riduce notevolmente il consumo di memoria [4].

La Figura B.1 mostra un grafo orientato con 5 vertici mentre la Figura B.2 mostra come la rete si possa rappresentare secondo le due tecniche appena viste.

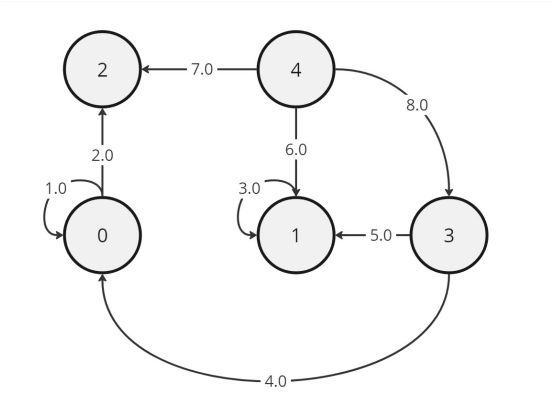

Figura B.1: Esempio di grafo diretto pesato

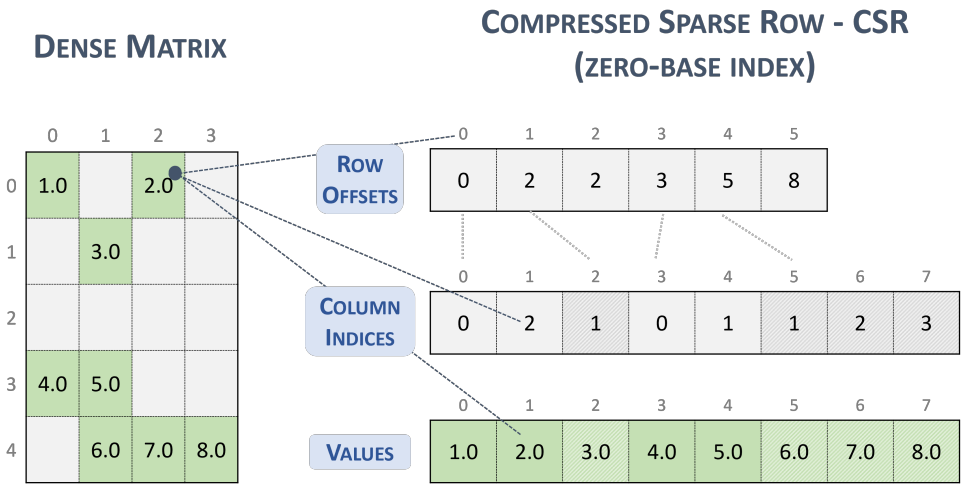

Figura B.2: Matrice di adiacenza vs CSR

Riferimenti

[1] Andrew V. Goldberg e Robert E. Tarjan. «A new approach to the maximum-flow problem». In: *J. ACM* 35.4 (ott. 1988), pp. 921–940. ISSN: 0004-5411. DOI: 10.1145/48014.61051.

[2] Chou-Ying Hsieh, Po-Chieh Lin e Sy-Yen Kuo. *Engineering A Workload-balanced Push-Relabel Algorithm for Massive Graphs on GPUs*. 2024. DOI: 10.48550/arXiv.2404.00270.

[3] G. Lancia. *A First Course in Operations Research: Lecture Notes for CS Students*. Independently Published, 2022.

[4] Nvidia. *Sparse Matrix Formats*. https://docs.nvidia.com/nvpl/_static/sparse/storage_format/sparse_matrix.html [Accessed: 10/10/2024]. 2024.

[5] Jiadong Wu, Zhengyu He e Bo Hong. «Chapter 5 - Efficient CUDA Algorithms for the Maximum Network Flow Problem». In: *GPU Computing Gems Jade Edition*. A cura di Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2012, pp. 55–66. ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00005-8.