Parallel breadth-first search

Simone Martucci - martucci.simone.91@gmail.com
Programmazione Multi Core e Many Core - A.A 2014/2015
Università di Roma Tor Vergata

31 maggio 2016

Sommario

La breadth-first search (**BFS**) è un algoritmo di ricerca su alberi o grafi, che partendo dal nodo radice, o da un qualsiasi nodo facente parte del grafo, permette di cercare il cammino, se esiste, fino ad un altro nodo scelto e connesso al nodo sorgente.
In questo lavoro la BFS è stata applicata per calcolare le distanze di ogni nodo appartenente al grafo dal nodo sorgente, cercando di parallelizzare l'algoritmo, così da ottenere prestazioni migliori.

1 Introduzione

L'obiettivo di questo lavoro è quello di illustrare diverse implementazioni dell'algoritmo della BFS. Per prima cosa verrà descritto il funzionamento della versione seriale di tale algoritmo per poi analizzare due implementazioni parallele dello stesso. Nel secondo capitolo è descritta l'implementazione di una versione che sfrutta le API di OpenMP. Nel capitolo successivo è invece descritta una possibile implementazione del medesimo algoritmo sull'architettura Cuda.

1.1 Breadth-First Search

Un grafo *G(v,e)* è composto da un insieme di vertici (v) e un insieme di archi (e), dato un nodo sorgente (s), appartenente al suddetto grafo, la *BFS* calcola le distanze di tutti i nodi raggiungibili da s, livello per livello.

Il procedimento da seguire per mettere in pratica tale ricerca è sintetizzato come segue:

1. Mettere in coda il nodo sorgente.
2. Togliere dalla coda un nodo (nella prima iterazione sarà il nodo sorgente) e marcarlo come visitato.
3. Selezionare i suoi vicini.
4. Per ogni vicino del nodo selezionato, se non ancora visitato, impostare la distanza dalla sorgente e metterlo in coda.
5. Se la coda è vuota, terminare, altrimenti ripetere il passo 2.

Nel listato successivo è presentato lo pseudo codice dell'algoritmo appena descritto:

Listing 1: Implementazione seriale

```
1  while( coda 1 non vuota ) {
2      for( nodo nella coda 1 ) {
3          // Prendo il nodo e lo rimuovo dalla coda 1
4          // Lo imposto come visitato
5          // Seleziono i suoi vicini
6          for ( vicino del nodo ) {
7              if( non ancora visitato ) {
8                  // Lo inserisco in coda 2
9                  // Imposto la distanza
10             }
11         }
12     }
13     // effettuo uno scambio delle code
14 }
```

1.2 Grafi

I grafi generati su cui si sono testati gli algoritmi descritti nei capitoli successvi sono essenzialmente di due tipi: i grafi del primo tipo sono generati in maniera casuale e, dato il metodo di generazione, sono grafi che non si avvicinano per caratteristiche a quelli utilizzati in casi di studio reali. I grafi del secondo tipo, invece, vengono generati tramite il metodo **R-MAT**, che ci permette di avere grafi più realistici, in quanto i nodi che lo compongono possono avere un grado molto differente tra di loro.

2 OpenMP

OpenMP (Open Multiprocessing) è un API multipiattaforma per la creazione di applicazioni parallele. É composta da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente che ne definiscono il funzionamento a run-time. Ogni direttiva ha come struttura la seguente:

$$\#\textbf{pragma omp} <\textit{direttiva}> \{...\}$$

e tutto il codice racchiuso nella sezione successiva farà parte del costrutto. In questo modo è possibile scrivere del codice "seriale" e sarà compito del compilatore renderlo parallelo.

Le direttive utilizzate per rendere il codice della BFS parallelo sono state le seguenti:

- **parallel for**: viene usata per rendere il ciclo definito subito dopo parallelo;
- **private (var ...)**: grazie a tale direttiva ogni thread possiede una sua copia delle variabili che vengono elencate all'interno delle parentesi tonde;

- **reduction (op: var...)**: sulle variabili elencate vengono effettuate le operazioni indicate in maniera efficiente;
- **critical**: il codice presente all'interno di tale sezione viene eseguito da un thread per volta;

2.1 Algoritmo

L'algoritmo sviluppato risulta molto simile a quello usato per la BFS seriale, la principale differenza tra i due è che in questo caso ogni nodo che viene estratto dalla coda viene elaborato da un thread differente, ciò permette di velocizzare l'elaborazione. Vi è però una considerazione da fare, ossia che miglioramenti prestazionali maggiori si ottengono quando il grado dei nodi risulta essere simile, poichè si ha che l'esplorazione dei vicini viene effettuata in maniera seriale da ogni thread, e quindi si avranno di certo risultati migliori su grafi del primo tipo rispetto a quelli generati con R-MAT.

Vediamo l'algoritmo per la BFS parallela con OpenMP nel dettaglio

Listing 2: Implementazione OpenMP

```
1  while( coda non vuota ) {
2      #pragma omp parallel for private(...) reduction(...)
3      for( nodo nella coda ) {
4          // Tolgo dalla coda
5          // Seleziono i suoi vicini
6          for (vicino del nodo) {
7              // se non ancora visitato
8              // imposto come visitato
9              // imposto la distanza
10             #pragma omp critical
11             {
12                 // inserisco il
13                 // nodo nella coda
14             }
15         }
16     }
17 }
```

Nel listing 2 sono indicate le operazioni fondamentali eseguite dall'algoritmo realizzato. Il *while* consente di scansionare il grafo per livelli e termina quando sono stati esplorati tutti i nodi raggiungibili a partire dalla sorgente S. Successivamente viene creata la regione parallela, istanziati i thread e vengono settate come private le variabili sulle quali i singoli thread andranno ad agire, infine grazie alla direttiva *for* viene scansionata la frontiera in modo parallelo.

Il ciclo più interno realizza la parte seriale dell'algoritmo, ovvero la visita dei vicini del singolo nodo. Nella sezione seriale infatti, ogni thread visita i vicini del nodo a lui assegnato e, se non già precedentemente visitati, li inserisce nella coda. L'algoritmo termina quando la coda è vuota.

Rispetto al codice seriale sono state apportate piccole modifiche che mirano ad ottimizzare l'algoritmo, ad esempio nel caso seriale per tener traccia dei nodi visitati abbiamo un array di *interi* mentre nel caso parallelo si è preferito utilizzare un array di *caratteri*, questo garantisce uno spreco di memoria minore in quanto nello spazio occupato da un intero possiamo allocare quattro caratteri. Oltre al puro spreco di memoria c'è sicuramente un guadagno prestazionale dovuto al fatto che l'array dei nodi visitati è acceduto frequentemente e utilizzare dei caratteri ci permette di tenere nella cache del processore più elementi, e quindi avere un esecuzione più veloce.

In secondo luogo il codice seriale presentava due code, una veniva utilizzata per leggere i nodi dell'attuale iterazione e l'altra invece per salvare quelli dell'iterazione successiva, nel nostro caso vi è un'unica coda, risparmiando così memoria, e utilizzando la direttiva *critical* ogni thread scrive su locazioni indipendenti.

2.2 Test e Risultati

I test sono stati eseguiti su un hardware tale da poter sfruttare un elevato parallelismo. Nel caso specifico si è utilizzata una macchina con un processore **Intel(R) Xeon(R) CPU E5645 v2 @ 2.40GHz** con 24 core e un elevato quantitativo di ram, in modo da poter lavorare con grafi di dimensione considerevole.

I risultati dei test vogliono mettere in luce lo *speed-up* relativo all'aumento dei thread utilizzati per esplorare il grafo. Tale speed-up è calcolato come il rapporto tra il tempo di esecuzione seriale e quello di esecuzione parallela.

Nella sezione successiva osserveremo dai grafici come effettivamente all'aumentare dei thread istanziati, lo speed-up cresca se il grafo risulta essere abbastanza grande da utilizzare tutte le risorse allocate, fino a raggiungere dei punti di saturazione.

Potremmo inoltre vedere che per i grafi del primo tipo lo speed-up maggiore rispetto a quelli generati con R-MAT, come già accennato precedentemente, poichè grafi generati con questo metodo sono molto sbilanciati ed è quindi possibile che un thread abbia molto più lavoro degli altri, portando di conseguenza alcuni thread a rimanere in attesa senza poter effettuare lavoro.

2.3 Grafici

Nei grafici mostrati a seguire si possono osservare i risultati ottenuti dai due tipi di grafi al crescere degli stessi in maniera differente. In un caso infatti il fattore di *scale* è fissato e aumenta il fattore di *edge*, nel secondo caso è fisso il fattore di *edge* e aumenta invece quello di *scale*.

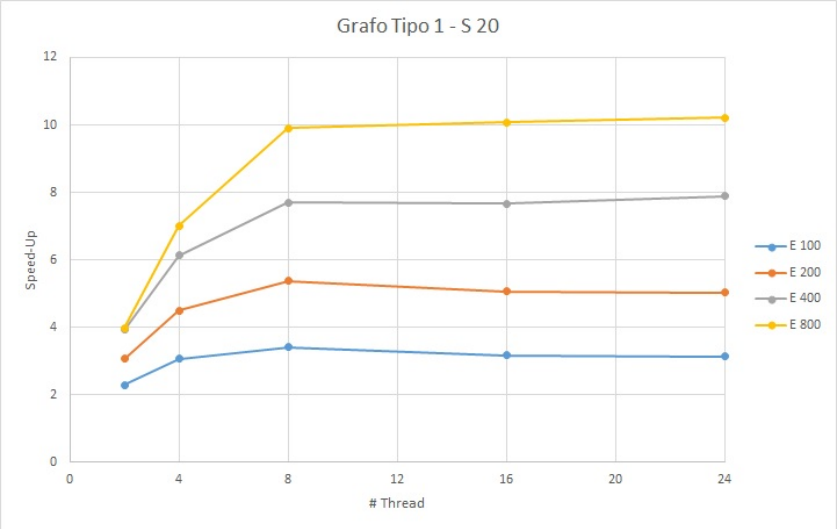

Figura 1: Speed Up al variare dei thread, con grafi casuali. Fattore di scale fisso a 20

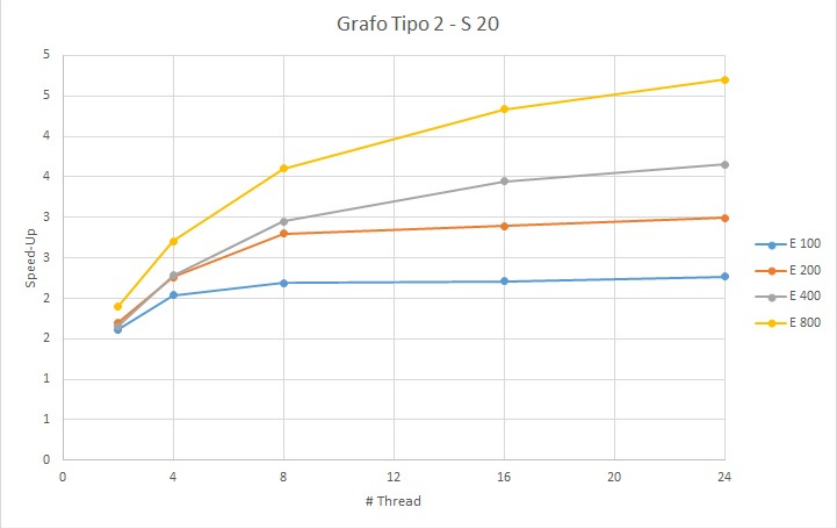

Figura 2: Speed Up al variare dei thread, con grafi di tipo R-MAT. Fattore di scale fisso a 20

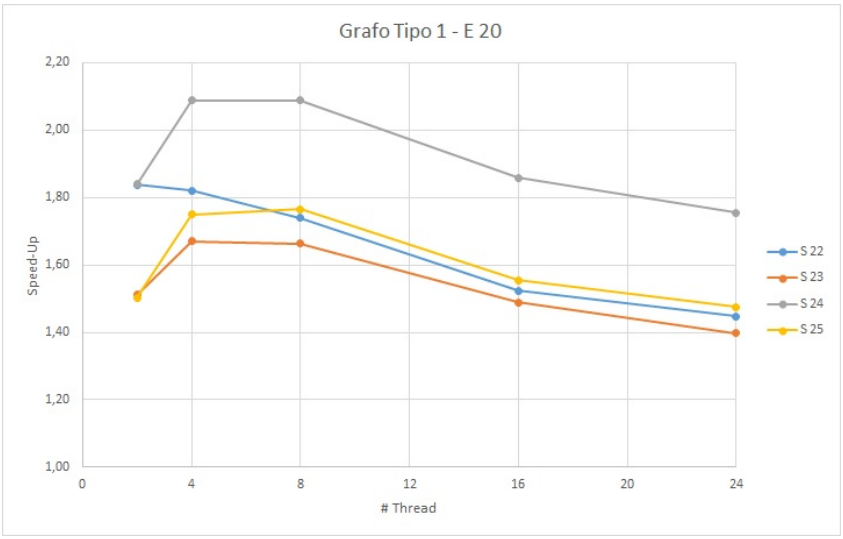

Figura 3: Speed Up al variare dei thread, con grafi casuali. Fattore di edge fisso a 20

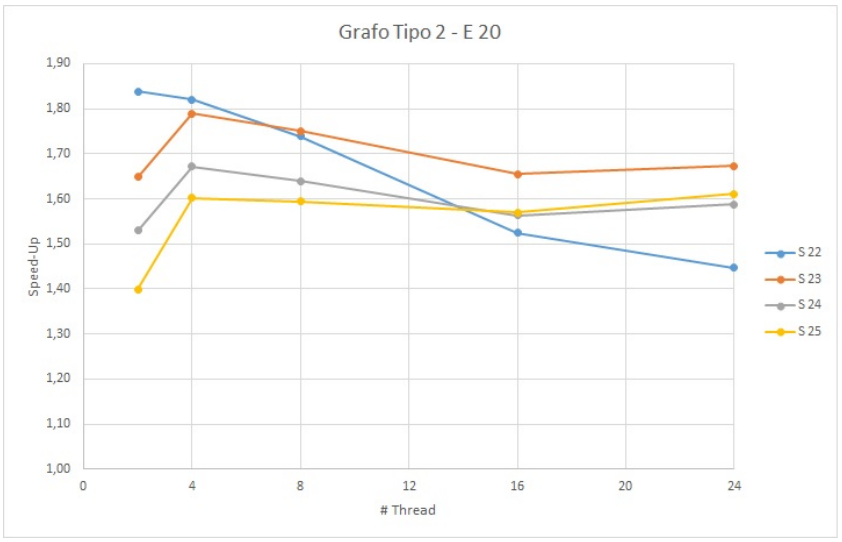

Figura 4: Speed Up al variare dei thread, con grafi di tipo R-MAT. Fattore di edge fisso a 20

3 CUDA

CUDA (Coompute Unified Device Architecture) è un'estensione del linguaggio C sviluppata da Nvidia che fornisce un'interfaccia di programmazione per GPU (Graphic Processor Unit). La CPU (Host) è responsabile dell'avvio del programma principale e dell'esecuzione del codice seriale, delegando l'elaborazione parallela alla GPU (Device). La programmazione CUDA richiede la definizione di funzioni C note chiamate *kernel*, le quali vengono eseguite in parallelo da una serie di thread sul device. I thread sono identificati tramite un id univoco, ed ognuno di essi esegue concorrentemente lo stesso kernel. Un kernel è eseguito in una griglia tridimensionale di blocchi di thread. I thread sono divisi in blocchi tridimensionali, thread dello stesso blocco condividono efficientemente una porzione di memoria (fast shared memory), e sono sincronizzati all'interno dello stesso, attraverso barriere hardware. Thread di blocchi differenti non possono sincronizzarsi, o condividere memoria.
I blocchi di thread si suddividono in gruppi chiamati *warp* e rappresentano l'unità minima schedulabile dalla GPU. Ogni thread in un warp esegue la stessa operazione su dati diversi simultaneamente agli altri thread del warp realizzando l'architettura SIMD. Al fine di sfruttare al meglio il parallelismo offerto da una GPU tutti i thread dello stesso warp dovrebbero eseguire lo stesso flusso di istruzioni, altrimenti si verificherebbe il fenomeno della divergenza. I thread del warp che divergono non vengono più eseguiti in parallelo, in altre parole eseguono in serie le operazioni, producendo quindi un degradamento delle prestazioni.

3.1 Algoritmo

In questa versione dell'algoritmo abbiamo in un primo momento l'allocazione della memoria e la copia dei dati necessari alla corretta esecuzione, compito che viene svolto in maniera seriale dalla CPU, successivamente l'host prosegue lanciando i kernel e quindi il device si occuperà dell'esecuzione vera e propria dell'algoritmo.

Per quanto riguarda l'allocazione dei dati viene creata una struttura apposita, atta a contenere tutte le informazioni necessarie per una corretta esecuzione, è possibile vedere tale struttura di seguito:

Listing 3: Struttura dati per CUDA

```
typedef struct _gpudata {
    char *redo;
    char *queue;
    char *frontier;
    int warp_size;
    UL *dist;
    UL vertex;
    UL level;
} gpudata;
```

Come possiamo osservare è presente una variabile *redo* che viene impostata ad uno se è necessaria alemno un ulteriore iterazione per portare a termine l'algoritmo, gli array *queue* e *frontier* sono necessari per tenere traccia rispettivamente

dei nodi da esplorare in questa iterazione e di tutti i loro vicini.
Le altre variabili sono il numero di thread contenuti in un singolo warp (*warp_size*), il numero di vertici del grafo (*vertex*), l'array delle distanze da calcolare (*dist*) e infine a che "livello" del grafo siamo arrivati con l'esplorazione (*level*), ossia la distanza del nodo dalla sorgente.

Passiamo ora ad analizzare come si è scelto di implementare l'algoritmo della bfs su un'architettura come questa.

Listing 4: Struttura dati per CUDA

```
1  __global__ void set_frontier(gpudata data, csrdata csrg) {
2
3      // Calcolo l'id del Warp
4      id = blockid * warps_block + threadid / warp_size;
5      while ( id < numero nodi ) {
6          if ( i-esimo nodo in coda ) {
7              // Rimuovo dalla coda
8              // Seleziono tutti i suoi vicini
9              for ( ogni vicino ) {
10                 /* Ogni thread del warp *
11                  * inserisce un vicino  *
12                  * nella frontiera      */
13             }
14         }
15         id += (gridDim * blockDim)/warp_size;
16     }
17 }
18
19 __global__ void compute_distance(gpudata data) {
20
21     id = (blockIdx.x*blockDim.x)+threadIdx.x;
22     // Controllo che l'id sia minore del numero di nodi
23     while (id < numero nodi) {
24         if ( nodo con indice tid sta nella frontiera ) {
25             // Devo fare almeno un'altra iterazione
26             // Lo rimuovo dalla frontiera
27             // Faccio un controllo per vedere se visitato
28             // Se non era stato visitato lo metto in coda
29             // E calcolo la distanza
30         }
31         // Incremento l'id del thread
32         tid += gridDim.x*blockDim.x;
33     }
34 }
```

Nel primo kernel, come possiamo vedere dal listato 4, facciamo in modo che ogni warp analizzi un nodo, se tale nodo è presente in coda per prima cosa viene rimosso e poi vengono selezionati tutti i suoi vicini, ognuno dei quali viene inserito nella frontiera da un thread differente facente parte del warp. Nel secondo

kernel invece viene analizzato ogni nodo da un thread differente, se questo è presente nella frontiera eseguo dei controlli e nel caso la distanza sia cambiata lo inserisco in coda, per eseguire il controllo dei suoi vicini nel primo kernel.

La scelta di utilizzare due kernel è stata imposta dal fatto che non esiste una primitiva di sincronizzazione tra i thread di tutti i blocchi, ma data la natura dell'algoritmo era necessario attendere il completamento della prima parte su ogni blocco. Data l'assenza di tale primitiva il modo migliore per implementarla è quello di lanciare due kernel in rapida successione, in quanto la CPU non attende il completamento del primo per lanciare il secondo, ma sarà la GPU ad accodarlo a quello già in esecuzione.

3.2 Test e Risultati

I test sono stati eseguiti su una GPU nVidia Titan con 6 GB di VRAM, 2688 cuda core. Nella sezione successiva osserveremo dai grafici come effettivamente all'aumentare delle risorse istanziate, lo speed-up cresca se il grafo risulta essere abbastanza grande da utilizzare tutte le risorse allocate, fino a raggiungere dei punti di saturazione.

Potremo inoltre vedere che per i grafi del primo tipo lo speed-up è molto maggiore rispetto a quelli generati con R-MAT, questo è dovuto al fatto che i grafi generati con tale metodo sono molto sbilanciati ed è quindi possibile che un thread abbia molto più lavoro degli altri, portando di conseguenza lo speed-up ad assottigliarsi.

Per grafi del primo tipo riusciamo ad ottenere dei valori di speed-up che oscillano tra 40 e 60 a seconda di come è costruito il grafo. Per grafi del secondo tipo invece si hanno speed-up minori compresi tra circa 30 e 40.

Si può osservare inoltre che utilizzare blocchi con troppi o troppo pochi thread introduce un peggioramento delle prestazioni, dai grafici infatti possiamo vedere che i migliori risultati si ottengono ponendo 128, 256 o 512 thread per ciascun blocco.

3.3 Grafici

Nei grafici mostrati a seguire si possono osservare i risultati trattati nel paragrafo precedente.

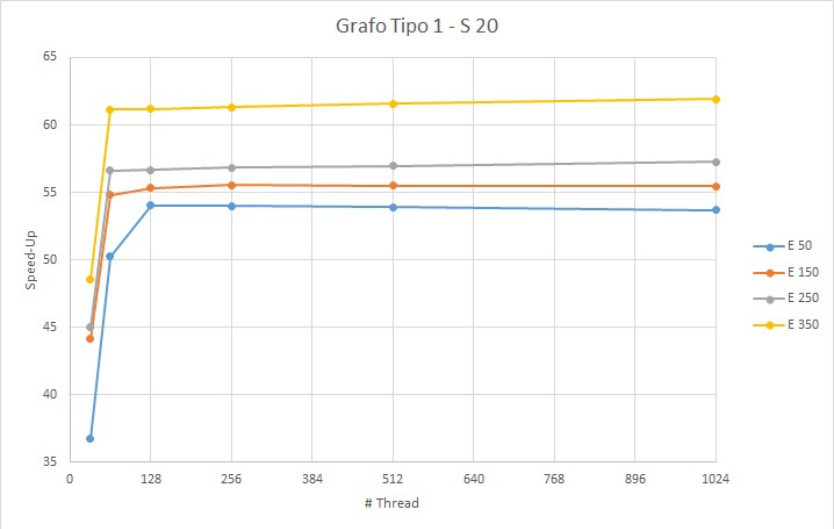

Figura 5: Speed Up al variare dei thread, con grafi casuali. Fattore di scale fisso a 20

Figura 6: Speed Up al variare dei thread, con grafi di tipo R-MAT. Fattore di scale fisso a 20

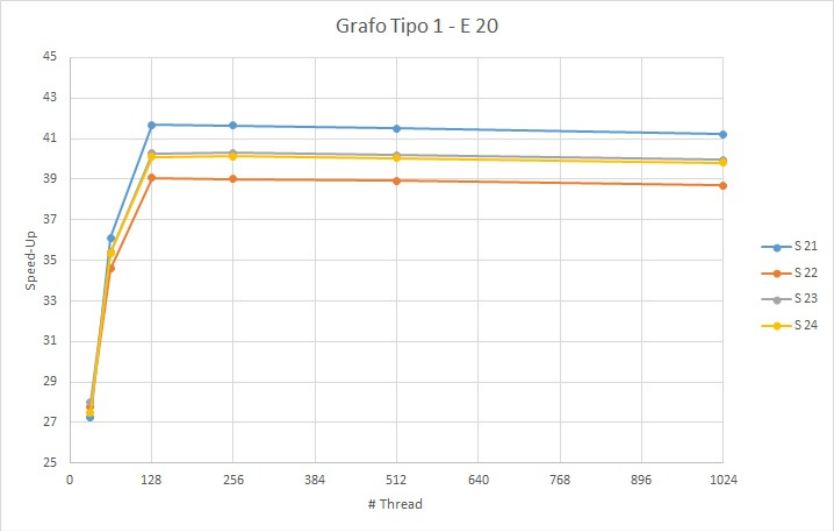

Figura 7: Speed Up al variare dei thread, con grafi casuali. Fattore di edge fisso a 20

Figura 8: Speed Up al variare dei thread, con grafi di tipo R-MAT. Fattore di edge fisso a 20