Parallel Computing
Relazione sull'implementazione di un Render di figure semitrasparenti

Lorenzo Mandelli

2019-2020

1 Abstract

Il presente elaborato si concentra sul rendering di figure nello spazio e analizza l'incremento di performance che si può ottenere con l'introduzione della programmazione parallela nei linguaggi c++ e java.

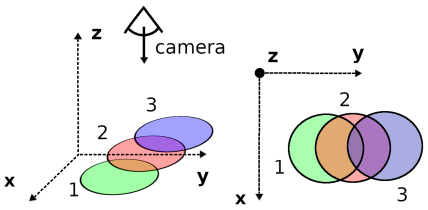

Figura 1: **Visione tridimensionale della struttura del problema.**

2 Formulazione del problema

Si supponga voler di renderizzare un insieme di N cerchi semitrasparenti, ciascuno caratterizzato da una posizione nello spazio data dalle coordinate (x,y,z), da un raggio R e da un colore C espresso in formato RGB con trasparenza $Alfa$. Si supponga inoltre che i cerchi siano disposti parallelamente al piano xy e che l'osservatore (la camera) del quale si vuole renderizzare la vista sia posto in cima all'asse z come mostrato in figura 1.

La disposizione dell'osservatore implica che l'ordine del rendering dei cerchi è individuato dalla posizione z: i cerchi con coordinata z più bassa dovranno essere i primi a essere renderizzati. É necessario che l'ordine sia mantenuto per ottenere un corretto risultato della sovrapposizione dei cerchi come mostrato nella seguente figura.

Figura 2: **Esempi di sovrapposizione corretta e non corretta.**

2.1 Versione sequenziale

L'algoritmo sequenziale si basa sulla realizzazione dei cerchi nell'ordine dato dalla coordinata z. La realizzazione è stata ottenuta attraverso le librerie Graphics2D in Java e Sfml in c++.
Di seguito sono mostrati i dettagli implementativi.

2.1.1 Implementazione

L'immagine è realizzata a partire da un array monodimensionale di lunghezza "$width * height * channels$", con $width$ e $height$ rispettivamente la larghezza e l'altezza dell'immagine e $channels = 4$ numero di canali corrispondenti ai canali RGB e trasparenza $Alpha$. Ogni elemento dell'array è inizialmente impostato al valore 255 che corrisponde a uno sfondo di colore bianco e con trasparenza nulla.
Successivamente per ogni cerchio in input al programma sono individuati gli elementi dell'array che corrispondono alle coordinate (x,y) dei pixel da realizzare. Questi elementi sono impostati al risultato dato dalla formula per la sovrapposizione in presenza di semitrasparenza:

$$e_i = C * (A) + e_i * (1 - A) \quad (1)$$

con e_i i-esimo elemento dell'array, C colore del cerchio corrente e A indice di semitrasparenza dai cerchi.
Infine dall'array è ricavata un immagine che è salvata con estensione *.png* attraverso le funzioni "$create$", "$saveToFile$" in Sfml e "$setRGB$", "$write$" in Graphics2D.

3 Versioni parallele

Sono state realizzate due versioni parallele del programma:

- La versione 1 si basa sul dividere l'immagine in sezioni e affidare la realizzazione di ciascuna di esse a un thread
- La versione 2 divide i cerchi da realizzare in un insieme di livelli che vengono realizzati in parallelo dai thread.

Entrambe le versioni prendono in ingresso il vettore contenente gli N cerchi ordinati rispetto alla coordinata z e producono in uscita l'immagine risultante dalla renderizzazione.
Le versioni sono state realizzate nei linguaggi di programmazione c++ e Java attraverso i threads della libreria standard e i Java Threads.

3.1 Versione 1

La prima versione si basa sulla divisione dell'immagine in T sezioni uguali con T numero di thread da utilizzare. Ogni thread possiede l'insieme totale dei cerchi e controlla per ciascuno di essi se i pixel specificati dalla loro posizione sono contenuti nella propria sezione. I pixel che risultano essere interni alla sezione vengono sovrapposti ai relativi pixel dell'immagine mediante

la formula di sovrapposizione dei colori in presenza di semitrasparenza (1).

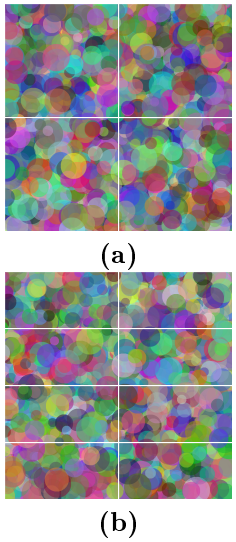

Figura 3: **Esempio di immagini con divisione a 4 sezioni (a) e a 8 sezioni (b).**

Di seguito sono riportati i dettagli implementativi.

3.1.1 Implementazione

La decisione su come dividere l'immagine in parti uguali avviene fattorizzando il numero di thread T in due interi a, b tali che $a*b$ =T: a rappresenta il numero di sezioni per ogni colonna dell'immagine e b il numero per ogni riga. Tra tutti i possibili fattori vengono scelti quelli che sono tra loro i meno distanti in modo da rendere le sezioni le più quadrate possibile. Ad esempio il numero T=4 può essere fattorizzato in (1,4),(2,2) e (4,1), ma dall'algoritmo è scelto (2,2) in modo da ottenere 4 quadrati invece di 4 rettangoli.
In seguito a partire dalla fattorizzazione sono calcolati i limiti di ogni sezione dell'immagine. Questi valori sono passati in ingresso ai rispettivi thread i quali operano in modo del tutto analogo alla versione sequenziale, trattata nella sezione 2.1, con la differenza che realizzano solamente i pixel dei cerchi che risultano essere interni ai limiti della propria sezione.

3.2 Versione 2

La seconda versione si basa sul parallelizzare le operazioni matematiche della sovrapposizione dei colori attraverso la divisione del vettore dei cerchi in T sottovettori con T numero di thread. Ogni thread esegue i calcoli necessari per la sovrapposizione dei cerchi inclusi nel proprio sottovettore e alla terminazione di tutti i thread le sovrapposizioni parziali ottenuti da questi vengono riunite per la corretta realizzazione dell'immagine.

3.2.1 Teoria

Si consideri un generico pixel p dell'immagine su cui devono essere sovrapposti N colori semitrasparenti. Indichiamo con C^k il k-esimo colore e sia $B = (1 - A)$ con A indice di semitrasparenza. Per la formula (1) il risultato della sovrapposizione dei N colori sarà dato da:

$$\begin{aligned} p =& (\{[(pB + AC^0)B + AC^1]B \\ &+ AC^2\}B + ...)B + AC^N \end{aligned}$$

ovvero:

$$\begin{aligned} p =& pB^{N-1} + AC^0B^{N-2} + ... \\ &+ AC^{N-1}B^1 + AC^NB^0 \end{aligned} \quad (1.5)$$

Indichiamo con $m1^t$ il risultato della sovrapposizione dei cerchi del sottovettore assegnato al thread t si ha che:

$$\begin{aligned} m1^t =& AC^0B^{m2^t} + AC^1B^{m2^t-1} + ... \\ &+ AC^{m2^t-1}B^1 + AC^{m2^t}B^0 \end{aligned}$$

con $m2^t$ numero di cerchi del sottovettore del thread t. Raccogliendo gli $m1^t$ nella formula (1.5) si ottiene:

$$p = \sum_k m1^k B^{Z_k} \tag{2}$$

$$Z_k = \sum_{t=k+1}^{T} m2^t \tag{3}$$

L'idea della versione 2 del programma è di far calcolare parallelamente ai thread i valori $m1^k$ e $m2^k$ di ogni pixel a partire dai cerchi contenuti nel proprio sottovettore per poi ottenere i valori dell'immagine attraverso la formula (2).

3.2.2 Esempio

Si considerino i parametri $N = 4$ numero di cerchi, $T = 2$ numero di thread, $A = 0.6$ indice di semitrasparenza e $p_{i,j}$ il pixel dell'immagine di cordinate (i, j) inizialmente impostato al colore RGB bianco $(255, 255, 255)$. Siano inoltre $\{C_1, C_2, C_3, C_4\}$ i cerchi con valori RGB rispettivamente: $\{(200, 0, 0), (100, 0, 0), (50, 0, 0), (80, 0, 0)\}$. Considerando la componente rossa $p^R_{i,j}$ del colore del pixel per la formula (1) di sovrapposizione del colore in presenza di semitrasparenza si avrà:

$$\begin{aligned} p^R_{i,j} =& (((255 * 0.4 + 200 * 0.6)0.4 + \\ &+ 100 * 0.6)0.4 \\ &+ 50 * 0.6)0.4 + 80 * 0.6 \\ &= 83.8 \end{aligned}$$

Siano $\{C1, C2\}$ assegnati al primo thread $t1$ e $\{C3, C4\}$ assegnati al secondo thread $t2$. I thread calcolano in parallelo i valori:

$$m1^{t1}_{i,j} = 200 * 0.4 * 0.6 + 100 * A = 108$$

$$m1^{t2}_{i,j} = 50 * 0.4 * 0.6 + 80 * A = 60$$

$$m2^{t1}_{i,j} = m2^{t2}_{i,j} = 2$$

Alla terminazione dei thread il programma principale esegue la formula (2) ottenendo il corretto risultato:

$$\begin{aligned} p^R_{i,j} &= 255(0.4)^4 + 108(0.4)^2 + 60 \\ &= 83.8 \end{aligned}$$

3.2.3 Implementazione

In questa sezione sono riportati i dettagli implementativi della versione 2.
Il programma istanzia T thread per computare in parallelo la sovrapposizione dei cerchi.
A ogni thread sono affidati: il sottovettore ottenuto partizionando il vettore di cerchi in ingresso in T parti e due array M1 e M2 di lunghezza "$width * height * channels$", con $width$ e $height$ rispettivamente la larghezza e l'altezza dell'immagine e $channels = 4$ numero di canali corrispondenti ai canali RGB e trasparenza $Alpha$. L'array M1 è utilizzato dal thread per la sovrapposizione dei cerchi appartenenti al proprio sottovettore in modo del tutto analogo alla versione sequenziale trattata nella sezione 2.1. L'array M2 è invece utilizzato per tenere traccia del numero di sovrapposizione fatte in ciascun pixel: ogni volta che viene aggiornato un elemento di M1 il corrispondente elemento di M2 viene incrementato di 1. Alla terminazione di ogni thread gli array M1 e M2 di ciascuno di essi vengono utilizzati per ricavare le componenti RGB di ogni pixel dell'immagine attraverso la formula (2) secondo il metodo trattato nella sezione 3.2.1

4 Risultati

I test operano la media di 10 esperimenti su dati generati randomicamente che vengono salvati su file di testo e letti dalle varie versioni dell'algoritmo. Gli speedup delle versioni parallele in c++ e Java sono stati calcolati relativamente alle versioni sequenziali nello stesso linguaggio.
I test sono stati eseguiti su un computer con processore Intel(R) Core(TM) i5-8250U con 4 core fisici e 8 core logici, 8.00 GB di RAM.
Di seguito indichiamo con N il numero di cerchi, con T il numero di thread, con W la larghezza dell'immagine e con H la sua altezza.
Nelle figure 4 e 5 sono mostrati gli speedup ottenuti al variare del numero di cerchi N e del numero di thread T. Come si può vedere gli speedup della versione 1 del programma sono maggiori rispetto a quelli della versione 2 in tutte le condizioni. Ciò è attribuibile alla necessità di gestire un maggiore numero di matrici di grandi dimensioni nella versione 2 rispetto che nella 1. Infatti se nella seconda versione del programma ci sono 2 matrici ausiliarie di dimensione pari a quella dell'immagine per ogni thread, nella prima versione ve ne è presente complessivamente una soltanto. Per questo stesso motivo la versione 2 del programma risente maggiormente dell'eccessivo aumento di thread come mostrato in figura 5.

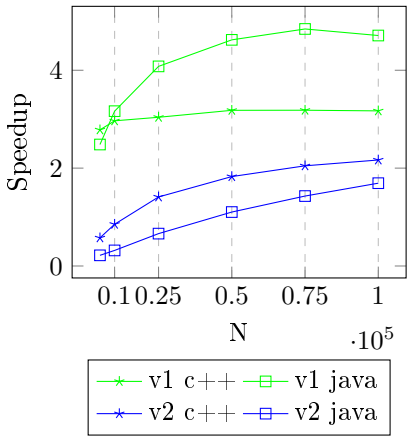

Figura 4: **Confronto degli speedup delle 2 versioni parallele dell'algoritmo in c++ e Java al variare di N con T=8.**

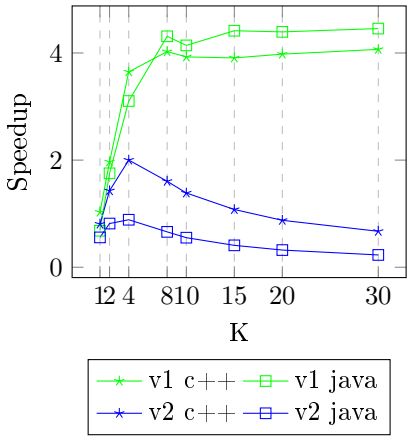

Figura 5: **Confronto degli speedup delle 2 versioni dell'algoritmo in c++ e java al variare di T con N= $2.5 * 10^5$.**

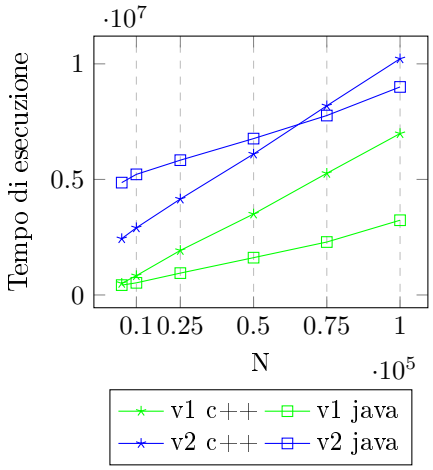

Figura 6: **Confronto dei tempi di esecuzione delle 2 versioni dell'algoritmo in c++ e java al variare di T con N= $2.5 * 10^5$.**

É stato possibile notare inoltre come entrambe le versioni dipendono dalla dimensione dell'immagine come mostrato in figura 7. All'aumentare dell'altezza e della larghezza dell'immagine si è riscrontato una diminuzione degli speedup.

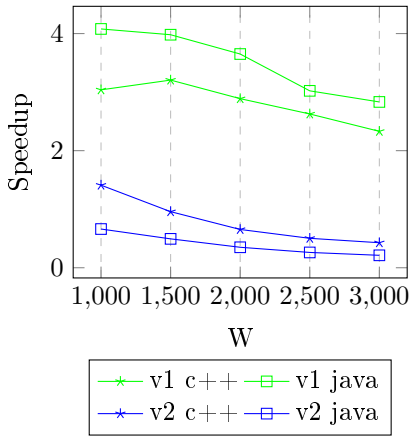

Figura 7: **Confronto degli speedup delle 2 versioni dell'algoritmo in c++ e java al variare delle dimensioni dell'immagine WxH con W = H, N= $2.5 * 10^5$, T=8.**

Di seguito nelle figure 8 e 9 sono mostrati due esempi di rendering con N pari a $5 * 10^4$ e $5 * 10^5$.

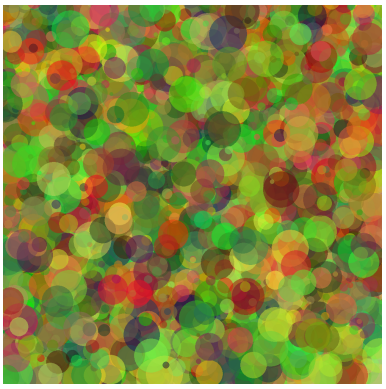

Figura 8: **Rendering ottenuto con N=$5 * 10^4$.**

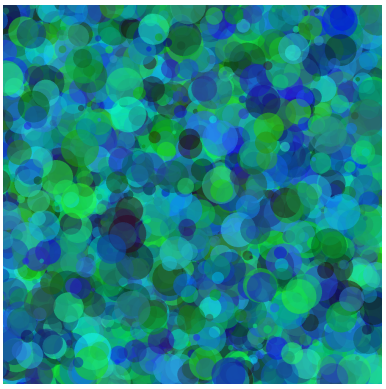

Figura 9: **Rendering ottenuto con N=$5 * 10^5$.**