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.

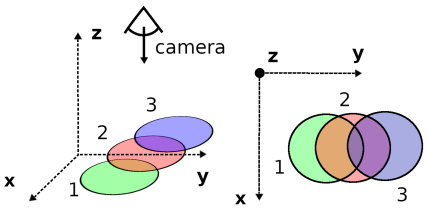


Figura 1: **Visione tridimensionale della struttura del problema.**

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.**

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++: per ogni cerchio i corrispondenti pixel dell'immagine sono impostati ai valori RGB e trasparenza $Alfa$ dati dal colore C.

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.

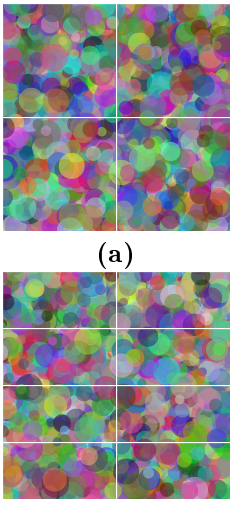
Entrambe le versioni prendono in ingresso il vettore contenente gli N cerchi ordinati rispetto alla coordinata z e producono in uscita l'immagine risultante dalla renderizzazione.
Le versioni sono state realizzate nei linguaggi di programmazione c++ e Java attraverso i threads della libreria standard 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 sua posizione sono contenuti nella propria sezione. I pixel che risultano essere interni alla sezione vengono sovrapposti ai relativi pixel dell'immagine, inizialmente impostata con uno sfondo bianco, mediante la formula di sovrapposizione dei colori in presenza di semitrasparenza:

$$p_{i,j} = C * (A) + p_{i,j} * (1 - A) \quad (1)$$

con $p_{i,j}$ pixel di coordinate (i, j), C colore del cerchio corrente e A indice di semitrasparenza dai cerchi.



(a)

(b)

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

3.2 Versione 2

La seconda versione opera nel modo seguente:

- Il vettore di cerchi ricevuto in ingresso è diviso in T sottovettori ciascuno dei quali è affidato a un thread. A ogni thread inoltre sono affidate due matrici M1 e M2 di dimensioni pari a quella dell'immagine: M1 per la sovrapposizione dei cerchi appartenenti al proprio sottovettore e M2 per il numero di sovrapposizione fatte in ciascun pixel.

- Ogni thread esegue la sovrapposizione dei colori dei cerchi del prorpio sottovettore mediante la formula (1) salvando i risultati nella sua matrice M1. Ogni volta che viene aggiornato un elemento di M1 il corrispondente elemento di M2 viene incrementato di 1.

- Alla terminazione di tutti i thread il pixel di coordinate (i,j) dell'immagine è ottenuto nel seguente modo:

$$p_{ij} = \sum_k m1_{ij}^k B^{Z_k} \qquad (2)$$

$$Z_k = \sum_{t=k+1}^{T} m2_{ij}^t \qquad (3)$$

  con p_{ij} pixel di coordinate (i,j) dell'immagine, $m1_{ij}^k$ e $m2_{ij}^k$ gli elementi di coordinate (i,j) delle matrici M1 e M2 del thread k, e $B = (1 - A)$ con A indice di semitrasparenza.

Per mostrare la validità delle formule (2) e (3) si consideri un generico pixel $p_{i,j}$ dell'immagine di coordinate (i, j) su cui devono essere sovrapposti N colori semitrasparenti. Indichiamo con $C_{i,j}^k$ il k-esimo colore del pixel e sia $B = (1 - A)$ con A indice di semitrasparenza. Per la formula (1) il risultato della sovrapposizione dei N colori sarà dato da:

$$\begin{aligned} p_{i,j} =& (\{[(p_{i,j}B + AC_{i,j}^0)B + AC_{i,j}^1]B \\ &+ AC_{i,j}^2\}B + ...)B + AC_{i,j}^N \end{aligned}$$

ovvero:

$$\begin{aligned} p_{i,j} =& p_{i,j}B^{N-1} + AC_{i,j}^0 B^{N-2} + ... \\ &+ AC_{i,j}^{N-1}B^1 + AC_{i,j}^N B^0 \qquad (4) \end{aligned}$$

Quindi se indichiamo con $m1_{ij}^t$ il risultato della sovrapposizione dei cerchi del sottovettore assegnato al thread t si ha che:

$$\begin{aligned} m1_{i,j}^t =& AC_{i,j}^0 B^S + AC_{i,j}^1 B^{S-1} + ... \\ &+ AC_{i,j}^{S-1}B^0 + AC_{i,j}^S B^0 \end{aligned}$$

con S numero di cerchi del sottovettore. Raccogliendo gli $m1_{i,j}^t$ nella formula (4) è possibile riportandosi alla formula (2).

3.2.1 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 $p_{i,j}^R$ 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):

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

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 mostrate 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.

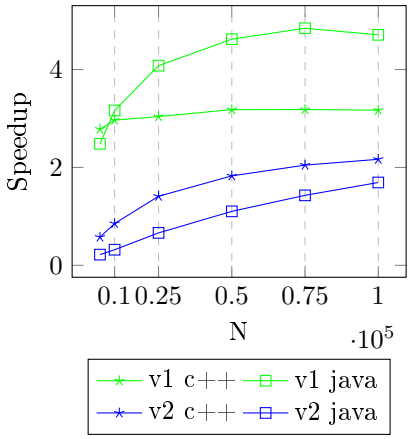


Figura 4: **Confronto degli speedup delle 2 versioni parallele dell'algoritmo in c++ e Java al variare di N con T=8.**

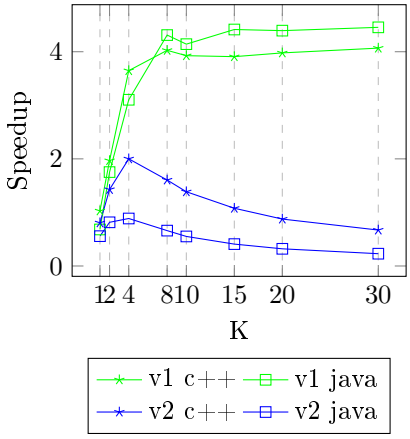


Figura 5: **Confronto degli speedup delle 2 versioni dell'algoritmo in c++ e java al variare di T con N= $2.5 * 10^5$.**

In figura 4 è mostrato il confronto dei tempi di esecuzione risultanti nelle varie versioni dell'algoritmo al variare del numero di oggetti N. Come si può vedere anche secondo questo aspetto la versione 1 risulta essere più performante della 2.

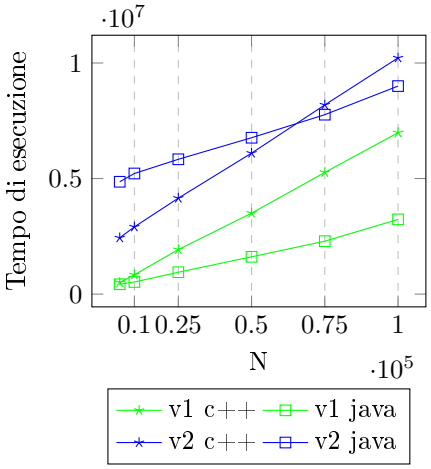


Figura 6: **Confronto degli speedup delle 2 versioni dell'algoritmo in c++ e java al variare di T con N= $2.5 * 10^5$.**

Entrambe le versioni inoltre dipendono dalla dimensione dell'immagine come è mostrato in figura 7. All'aumentare dell'altezza e della larghezza dell'immagine entrambi gli speedup diminuiscono.

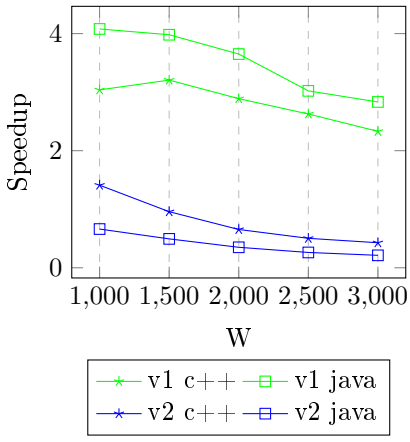


Figura 7: **Confronto degli speedup delle 2 versioni dell'algoritmo in c++ e java al variare delle dimensioni dell'immagine WxH con W = H, N= $2.5 * 10^5$, T=8.**

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

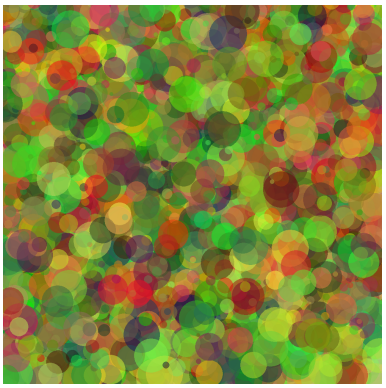


Figura 8: **Rendering ottenuto con N=$5 * 10^4$.**

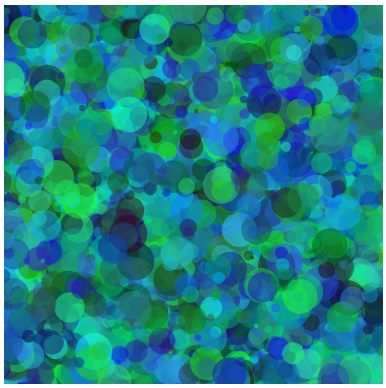


Figura 9: **Rendering ottenuto con N=$5 * 10^5$.**