Parallelizzazione del rendering di cerchi

Silvia Dani
Indirizzo e-mail
silvia.dani@stud.unifi.it

Niccolò Niccoli
Indirizzo e-mail
niccolo.niccoli@stud.unifi.it

Abstract

Questo esperimento vuole analizzare lo speed up che si ottiene parallelizzando del codice. Per fare ciò viene confrontata un'implementazione sequenziale del codice con una parallelizzata utilizando OpenMP. Il codice che viene parallelizzato si occupa del rendering di un numero predefinito di cerchi su un'immagine di dimensioni anch'esse definite a priori.

1. Introduzione

L'obiettivo di questo elaborato è quello di generare dei cerchi di diverso diametro e colore con una determinata trasparenza caratterizzati da coordinate x, y, z, renderizzarli e misurare il tempo impiegato a fare ciò, confrontando una versione sequenziale con una versione parallela e variando il numero di threads utilizzati. I cerchi vengono disegnati a partire da quello più in profondità fino ad arrivare a quello più in alto sovrapponendoli mano a mano e per questo motivo è fondamentale l'ordine in cui vengono aggiunti i cerchi. È stato deciso di rappresentare la coordinata z come l'ordinamento dei cerchi all'interno di un array.

2. Progettazione

Alla base è presente una struct Circle che contiene:

- raggio;
- punto centrale;
- array che contiene il valori del colore del cerchio diviso per canali (R, G e B).

L'approccio che viene utilizzato è quello di creare un array di cerchi e di popolarlo, in seguito lo si divide in sottogruppi e per ognuno su una matrice trasparente vengono disegnati in ordine i cerchi presenti in esso. Al termine di questo passaggio si hanno tante matrici quante le divisioni dell'array con disegnati dei cerchi semitrasparenti e sfondo trasparente. È stato poi implementato un metodo che permette di unire due immagini analizzandole prima pixel per pixel e poi fondendole. Questo metodo viene utilizzato per unire le immagini con sfondo trasparente create al passaggio prima con una matrice completamente bianca. Le immagini vengono fuse partendo da quella che contiene i cerchi "più profondi".

3. Parallelizzazione

Per effettuare la parallelizzazione si è deciso di utilizzare

```
#pragma omp parallel
#pragma omp for
```

per disegnare i cerchi sulle immagini parziali e

```
#pragma omp for
```

nel metodo per sovrapporre due immagini dividendole in chunck in quanto i thread erano stati già creati in precedenza.

Figure 1. sequenziale

Figure 2. parallelizzata 2 processori

Figure 3. parallelizzata 4 processori

Figure 4. parallelizzata 8 processori

```
#pragma omp parallel default(none) shared (
    images, circles, nCircles, numProcs,
    minNumCirclesPerImg, white, imageHeight,
     imageWidth)
#pragma omp for
   for (int i = 0; i < numProcs; i++) {
      cv::Mat background = cv::Mat(
          imageHeight, imageWidth, CV_8UC4,
          cv::Scalar(255, 255, 255, 0));
      for (int j = i*minNumCirclesPerImg; j
           < (i+1)*minNumCirclesPerImg; j++)
        {
      images[i].copyTo(background);
      cv::circle(images[i], circles[j].
          center, circles[j].radius, cv::
          Scalar(circles[j].color[0],
          circles[j].color[1], circles[j
          ].color[2], 255), -1);
      cv::addWeighted(images[i], ALPHA,
          background, 1.0 - ALPHA, 0.0,
          images[i]);
  }
```

```
    }

    int tileHeight = imageHeight / numProcs;

#pragma omp for
    for (int threadIdx = 0; threadIdx <
        numProcs; threadIdx++) {
       for (int i = 0; i < numProcs; i++) {
          overlayImage(&white, &images[i],
              tileHeight * threadIdx,
              tileHeight * (threadIdx + 1));
       }
    }
```

4. Analisi

La versione sequenziale e quella parallelizzata sono state messe a confronto ed è stato misurato il tempo necessario a renderizzare un'immagine al variare del numero di cerchi (200, 1000, 10000, 100000), della dimensione della matrice che si vuole riempire (256x256, 512x512, 1024x1024) e al numero dei processori coinvolti (2, 4, 8).

5. Risultati

Le immagini 1, 2, 3, 4 sono alcune di quelle ottenute come output, in questo caso si tratta del rendering di 200 cerchi su una matrice 256x256. Come si vede tutte le immagini sono identiche e quindi non vi è perdita di informazioni tra la versione sequenziale e quella parallelizzata.

Le immagini 5, 6, 7, 8 mostrano invece come varia l'output al variare del numero di cerchi disegnati. Le immagini scelte sono quelle con le dimensioni più grandi e, siccome è stato visto che non vi è differenza di output tra versione sequenziale e parallelizzata, viene mostrata solo un immagine per tipologia.

Nella tabella 1 (in appendice) sono visibili i tempi di esecuzione dei rendering. I tempi sono indicati in secondi. Al variare del numero di thread è possibile vedere che è stata ripetuta l'esecuzione della versione sequenziale. Questa scelta è stata fatta per poter stampare i dati in modo più organico. Siccome il codice sequenziale, per definizione, viene eseguito su di un singolo thread è possibile aspettarsi (e infatti è così) che i tempi siano tra loro simili.

Successivamente sono state confrontate le versioni parallele e la versione sequenziale a parità di dimensioni dell'immagine. Il tempo che è stato considerato per la versione sequenziale è la media tra i tempi che sono stati ottenuti e che sono stati riportati nella tabella 2.

Attraverso i grafici 9, 10, 11, 12, 13, 14 è immediato osservare che la versione parallela (anche quella che utilizza solo 2 threads) risulta molto conveniente quando il carico di lavoro diventa intenso. In particolare lo *speed up* massimo registrato rispetto alla versione sequenziale è: 1.96 quando si utilizzano 2 thread, 2.69 quando se ne utilizzano 4 e 4.37 nel caso di 8 thread.

6. Conclusioni

Dagli esperimenti svolti è possibile notare che la parallelizzazione del codice contribuisce in modo sensibile a ridurre il tempo di esecuzione. In particolare si può notare che lo *speed up* massimo si ottiene con 8 threads.

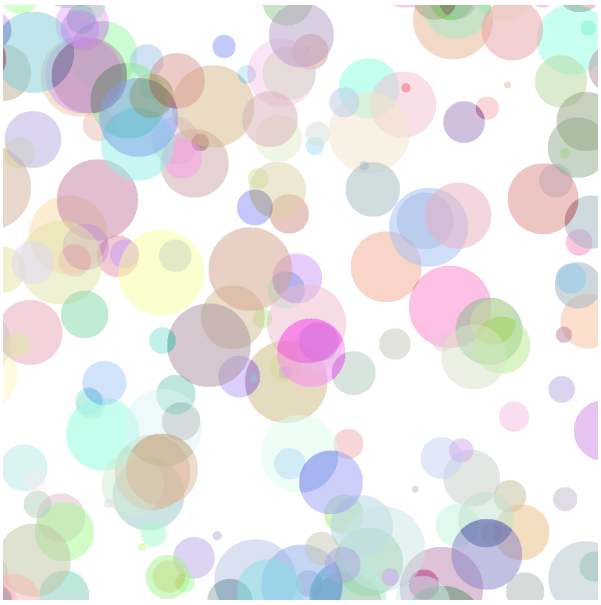

Figure 5. 200 cerchi

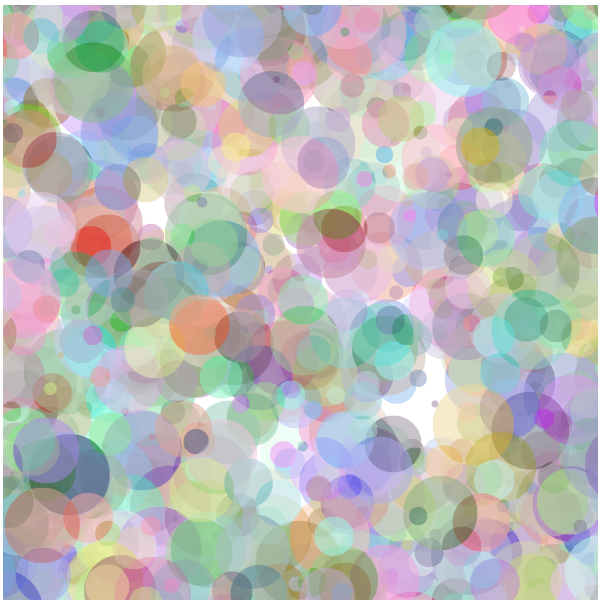

Figure 6. 1000 cerchi

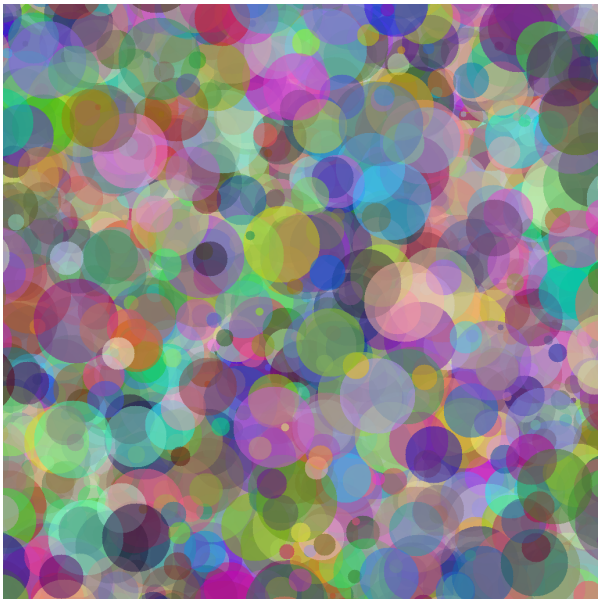

Figure 7. 10000 cerchi

Figure 8. 100000 cerchi

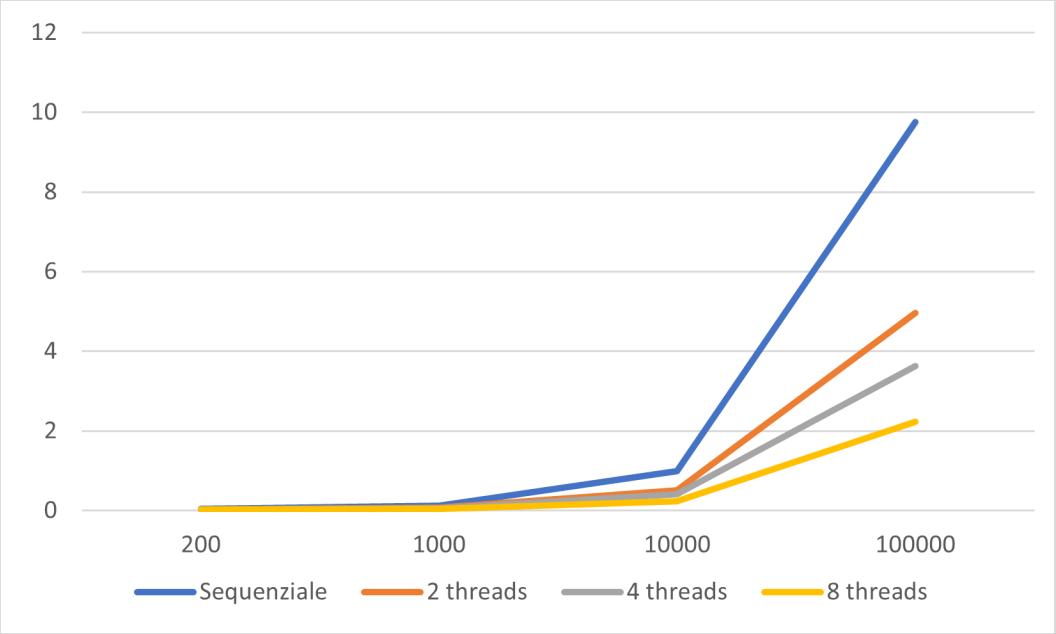

Figure 9. Confronto relativo a rendering immagine 256x256px.

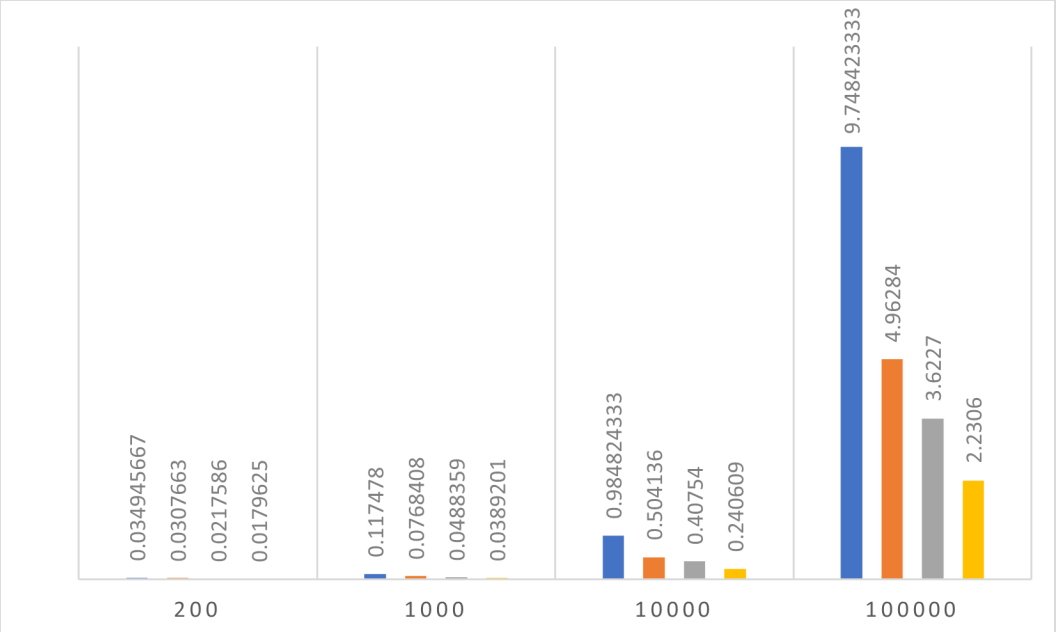

Figure 10. Confronto relativo a rendering immagine 256x256px.

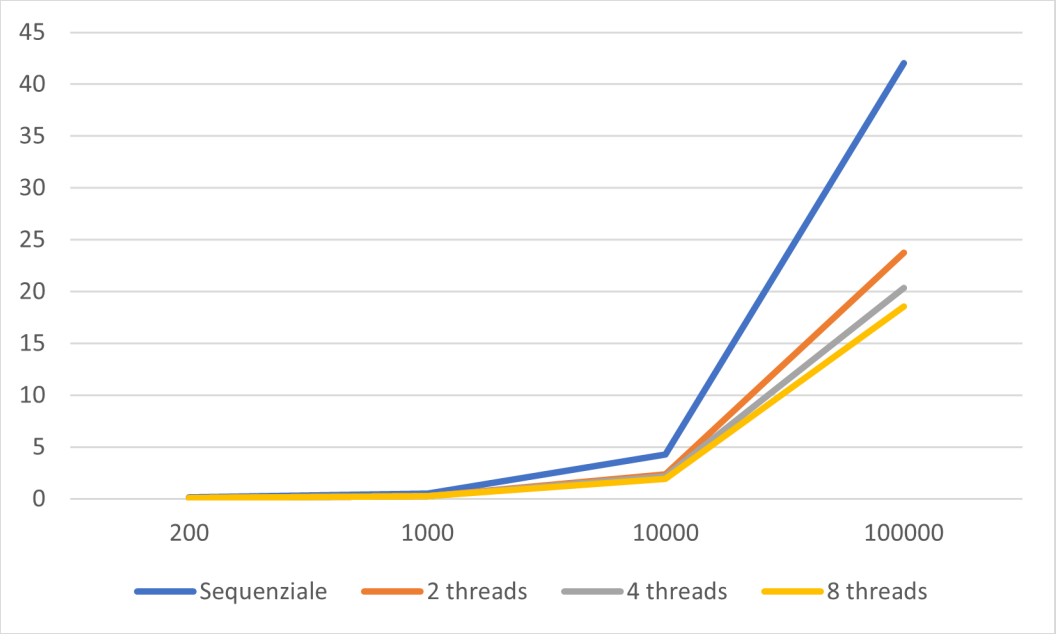

Figure 11. Confronto relativo a rendering immagine 512x512px.

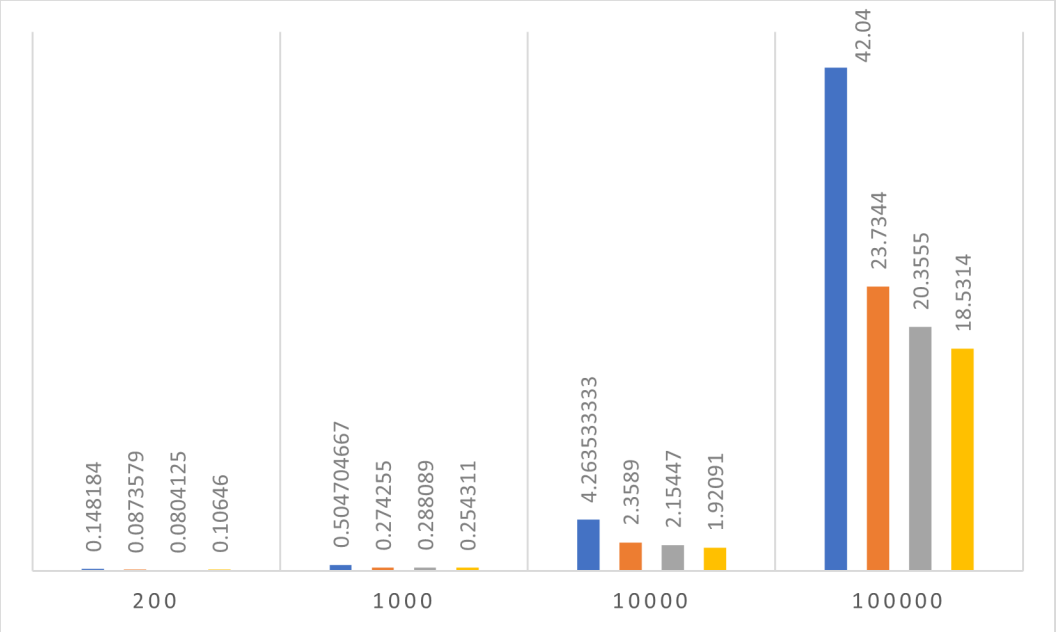

Figure 12. Confronto relativo a rendering immagine 512x512px.

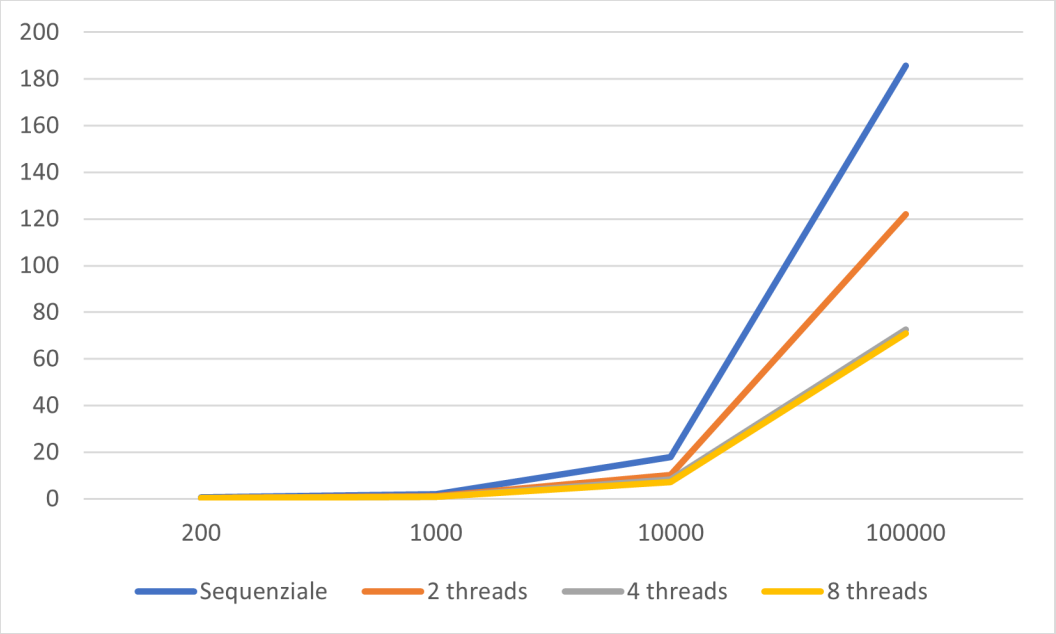

Figure 13. Confronto relativo a rendering immagine 1024x1024px.

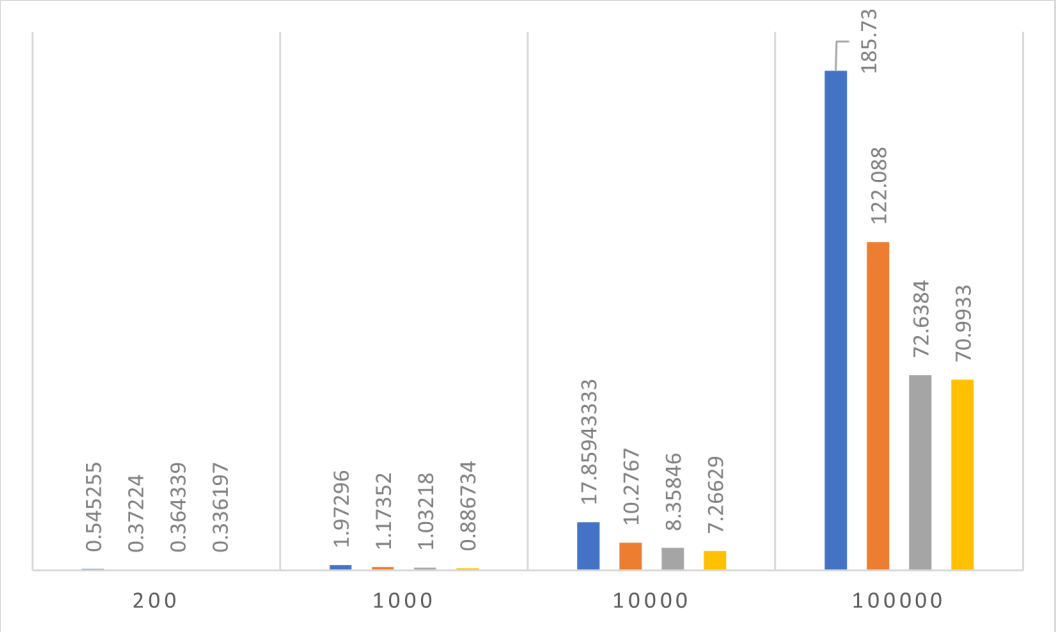

Figure 14. Confronto relativo a rendering immagine 1024x1024px.

7. Appendice

2 threads, 256x256 # cerchi	Versione sequenziale	Versione parallela
200	0.0431362	0.0307663
1000	0.129362	0.0768408
10000	0.968143	0.504136
100000	9.57979	4.96284
4 threads, 256x256 # cerchi		
200	0.146311	0.0873579
1000	0.465663	0.274255
10000	4.22227	2.3589
100000	41.1959	23.7344
2 threads, 1024x1024 # cerchi		
200	0.521502	0.37224
1000	1.93846	1.17352
10000	17.518	10.2767
100000	196.753	122.088
4 threads, 256x256 # cerchi		
200	0.0310891	0.0217586
1000	0.113947	0.0488359
10000	1.03351	0.40754
100000	10.2293	3.6227
4 threads, 512x512 # cerchi		
200	0.144339	0.0804125
1000	0.513008	0.288089
10000	4.40996	2.15447
100000	43.7238	20.3555
4 threads, 1024x1024 # cerchi		
200	0.547389	0.364339
1000	2.00842	1.03218
10000	18.6991	8.35846
100000	184.024	72.6384
8 threads, 256x256 # cerchi		
200	0.0306117	0.0179625
1000	0.109125	0.0389201
10000	0.95282	0.240609
100000	9.43618	2.2306
8 threads, 512x512 # cerchi		
200	0.153902	0.10646
1000	0.535443	0.254311
10000	4.15837	1.92091
100000	41.2082	18.5314
8 threads, 1024x1024 # cerchi		
200	0.566874	0.336197
1000	1.972	0.886734
10000	17.3612	7.26629
100000	176.424	70.9933

Table 1. Tempi di esecuzione.

256x256 # cerchi	Tempo medio versione sequenziale
200	0.034945667
1000	0.117478
10000	0.984824333
100000	9.748423333
512x512 # cerchi	
200	0.148184
1000	0.504704667
10000	4.263533333
100000	42.04263333
1024x1024 # cerchi	
200	0.545255
1000	1.97296
10000	17.85943333
100000	185.7336667

Table 2. Tempi medi di esecuzione del codice sequenziale.