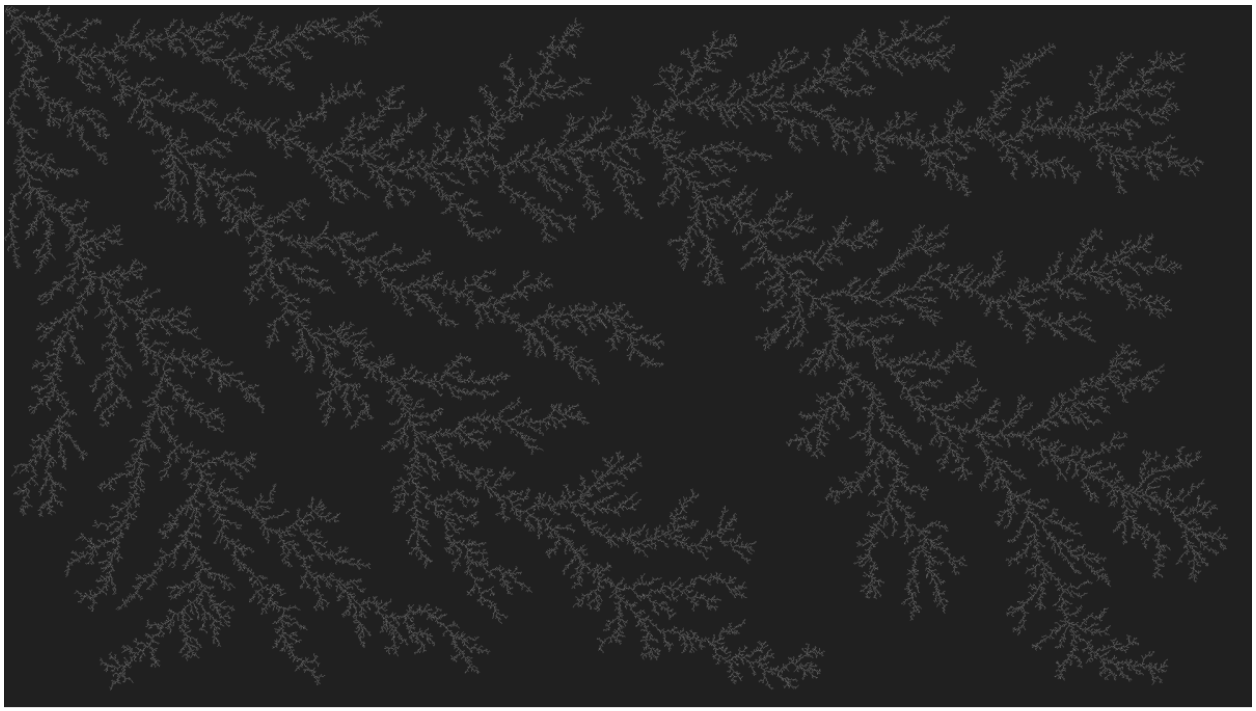


Immagine ottenuta dall' esecuzione dell'implementazione seriale

# Diffusion-limited aggregation

**3 aprile 2021**


**Studenti:** **Cristian Buciu 1871137**

**Matteo Catalano 1836454**


**Corso:** **Programmazione di Sistemi Multicore**

## PANORAMICA

Diffusion limited aggregation (DLA) è il processo per cui le particelle che subiscono un cammino casuale dovuto al moto browniano si cristallizzano per formare aggregati di tali particelle. Questa teoria è applicabile all'aggregazione in qualsiasi sistema in cui la diffusione è il mezzo primario di trasporto nel sistema. Il DLA può essere osservato in molti sistemi come l'elettrodeposizione, il flusso di Hele-Shaw, i depositi minerali e la rottura dielettrica.

## OBIETTIVI

L’obiettivo del progetto è quello di implementare un sistema che simuli il problema del Diffusion limited aggregation usando le due metodologie viste a lezione. Quindi abbiamo dovuto realizzare le seguenti implementazioni:

- implementazione che utilizza i Posix Thread (PThread)
- implementazione che utilizza OpenMP
- implementazione seriale

## SPECIFICHE

I parametri di base della simulazione sono la dimensione della griglia 2D, il numero iniziale di particelle, il numero di iterazioni e il "seme" cristallino iniziale

## TAPPE  INTERMEDIE

### Implementazione seriale

Nella fase iniziale del progetto ci siamo concentrati nel progettare una soluzione seriale per il problema.
Inizialmente vengono forniti in input: altezza della griglia 2D, larghezza della griglia 2D, numero iterazioni, numero particelle, coordinata x del cristallo iniziale e coordinata y del cristallo iniziale. Una volta presi in input questi valori, l’algoritmo genera una matrice di caratteri <space> dove viene marcato il cristallo iniziale mediante il carattere “o”. Una volta fatto ciò vengono generate le varie particelle sparse per tutta la griglia in modo casuale e memorizzate in un array. Una volta creato l’ambiente, inizia la simulazione dove, basandosi sui parametri forniti comincerà a riprodurre il comportamento nella griglia delle varie particelle. Cioè per ogni iterazione e per ogni particella , verrà assegnata una nuova posizione alla particelle e se nelle celle circostanti è presente un cristallo la particella si cristallizzerrà (la posizione della particella nella griglia viene

marcata con il carattere “o” ). Finita la simulazione, mediante i valori appena calcolati della matrice viene generato un file in formato .ppm che permette di visualizzare il risultato ottenuto.

```
pseudocodice:
inizializzo matrice
genero le particelle
per ogni iterazione:
    per ogni particella:
        sposta(particella)
        se la particella è vicina ad un cristallo
            cristallizza(particella)
            aggiorna matrice
creaimmagine(matrice)
```

## Implementazione con PThread (RW Lock / Barrier)

In questa implementazione abbiamo cercato di parallelizzare la soluzione precedentemente descritta.

- **RWLock:**
  In questa soluzione abbiamo pensato di dividere il carico di lavoro fra i vari thread. Ciascuno di essi crea una parte delle particelle e si occuperà del movimento e del controllo dei cristalli. La parte critica è l’accesso alla matrice condivisa fra tutti i thread sia in lettura per quando riguarda il controllo delle celle circostanti e sia in scrittura per quanto riguarda la “cristallizzazione” ovvero l’aggiornamento della cella. Per gestire l’accesso alla sezione critica abbiamo deciso di usare i read-write lock.
  Un read-write lock è una primitiva di sincronizzazione che consente l'accesso simultaneo per operazioni di sola lettura, mentre le operazioni di scrittura richiedono un accesso esclusivo. Ciò significa che più thread possono leggere i dati in parallelo ma un blocco esclusivo è necessario per scrivere o modificare i dati. [2]

```
pseudocodice:
inizializzo matrice
lancio i thread che lavorano sulla funzione f()
join()
creaimmagine(matrice)

f():
   calcolo il numero di particelle che devo gestire
   genero le mie  particelle
   per ogni iterazione:
     per ogni particella:
         sposta(particella)
         readLock()
         controllo se la particella è vicina ad un cristallo
         readUnlock()
         se il controllo precedente è vero
             cristallizza(particella)
             writeLock()
             aggiorna matrice
             writeUnlock()
```

- **Barrier**: In questa implementazione abbiamo spezzato l'intero algoritmo in due fasi che vengono ripetute ad ogni iterazione :
    - **Fase A** :  Vengono spostate tutte le particelle, viene eseguito il controllo sulla griglia e si memorizzano in una lista le particelle pronte a cristallizzarsi. Una volta fatto ciò i thread vengono messi in attesa che tutti gli altri abbiano finito questa fase.
    - **Fase B** : Utilizzando la lista generata in precedenza si aggiorna la griglia. Per sincronizzare i thread fra di loro si utilizza una seconda barrier.

```
pseudocodice:
inizializzo matrice
lancio i thread che lavorano sulla funzione f()
join()
creaimmagine(matrice)

f():
   calcolo il numero di particelle che devo gestire
   genero le mie particelle
   per ogni iterazione:
     per ogni particella:
          sposta(particella)
    per ogni particella:
          se la particella è vicina ad un cristallo
                lista_da_cristallizzare.add(particella)
    barrier()
    per ogni elemento p  in lista_da_cristallizzare:
          griglia.update(p)
    barrier()
```

## Implementazione che utilizza OpenMP con Barrier

In questa implementazione abbiamo riutilizzato l'algoritmo parallelo dei PThread con barrier definito in precedenza sfruttando le direttive di compilazione della libreria omp.h

## Valutazione delle prestazioni

Per misurare i tempi delle varie esecuzioni abbiamo preso il tempo di orologio a muro all'inizio del programma e il tempo alla fine e li abbiamo sottratti . Nelle tabelle di seguito abbiamo riportato tutti i dati raccolti

Simulazioni eseguite su un computer con wsl che ha il seguente processore:

https://ark.intel.com/content/www/it/it/ark/products/80817/intel-core-i5-4460-processor-6m-cache-up-to-3-40-ghz.html

per input piccolo (matrice 100x100 con 10.000 iterazioni e 1500 particelle)

| tipo/num thread | 1 thread | 2 thread | 3 thread | 4 thread |
|---|---|---|---|---|
| serial | 83.029 us | | | |
| PThread Barrier | | 573.995 us | 932.436 us | 1.253.870 us |
| Speedup | | 0,144 | 0,089 | 0,066 |
| Efficienza | | 0,072 | 0,029 | 0,165 |
| OpenMP | | 72.794 us | 71.245 us | 70.951 us |
| Speedup | | 1,14 | 1,16 | 1,17 |
| Efficienza | | 0,57 | 0,38 | 0,29 |
| PThread RWLock | | 102.389 us | 133.146 us | 157.297 us |
| Speedup | | 0,81 | 0,62 | 0,52 |
| Efficienza | | 0,40 | 0,20 | 0,13 |
| numeri di lock utilizzati | | rlock = 536.447<br>wlock = 1500 | rlock = 661.711<br>wlock = 1500 | rlock = 647.771<br>wlock = 1500 |

| tipo/num thread | 5 thread | 6 thread | 7 thread | 8 thread |
|---|---|---|---|---|
| serial | | | | |
| PThread Barrier | 1.580.767 us | 1.928.938 us | 2.258.932 us | 2.587.149 us |
| Speedup | 0,052 | 0,043 | 0,036 | 0,032 |
| Efficienza | 0,0104 | 0,0071 | 0,0051 | 0,004 |
| OpenMP | 1.579.902 us | 1.918.735 us | 2.255.809 us | 2.592.425 us |
| Speedup | 0,052 | 0,043 | 0,036 | 0,032 |
| Efficienza | 0,0104 | 0,0071 | 0,0051 | 0,004 |
| PThread RWLock | 162.945 us | 166.575 us | 177.464 us | 173.097 us |
| Speedup | 0,50 | 0,49 | 0,46 | 0,47 |
| Efficienza | 0,1 | 0,081 | 0,065 | 0,058 |
| numeri di lock utilizzati | rlock:712.854<br>wlock = 1500 | rlock = 746.560<br>wlock = 1500 | rlock = 787.851<br>wlock = 1500 | rlock = 952.657<br>wlock = 1500 |

per input grande (matrice 1000x1000 con 100.000 iterazioni e 100.000 particelle)

| tipo/num thread | 1 thread | 2 thread | 3 thread | 4 thread |
|---|---|---|---|---|
| seriale | 89.394.512 us | | | |
| PThread Barrier | | 45.875.677 us | 37.171.559 us | 35.235.034 us |
| Speedup | | 1.948 | 2.404 | 2.537 |
| Efficienza | | 0.974 | 0.801 | 0.634 |
| OpenMP | | 41.211.665 us | 28.469.103 us | 22.986.919 us |
| Speedup | | 2.169 | 3.140 | 3.88 |
| Efficienza | | 1 | 1 | 0.97 |
| PThread RWLock | | 142.844.314 us | 163.407.352 us | 180.390.314 us |
| Speedup | | 0.625 | 0.547 | 0.495 |
| Efficienza | | 0.312 | 0.182 | 0.12 |
| numero di lock utilizzati | | rlock = 1.451.481.869<br>wlock = 100.000 | rlock = 1.412.266.662<br>wlock = 100.000 | rlock= 1.428.009.036<br>wlock = 100.000 |

| tipo/num thread | 5 thread | 6 thread | 7 thread | 8 thread |
|---|---|---|---|---|
| seriale | | | | |
| PThread Barrier | 47.586.108 us | 44.597.267 us | 46.468.088 us | 48.590.883 us |
| Speedup | 1.878 | 2.004 | 1.923 | 1.835 |
| Efficienza | 0.375 | 0.334 | 0.274 | 0.229 |
| OpenMP | 47.356.920 us | 46.326.654 us | 46.241.154 us | 48.848.101 us |
| Speedup | 1.887 | 1.929 | 1.933 | 1.830 |
| Efficienza | 0.377 | 0.321 | 0.276 | 0.228 |
| PThread RWLock | 178.100.831 us | 180.151.598 us | 188.014.839 us | 201.335.163 us |
| Speedup | 0.501 | 0.496 | 0.475 | 0.444 |
| Efficienza | 0.1 | 0.082 | 0.067 | 0.055 |
| numero di lock utilizzati | rlock = 1.660.884.752<br>wlock=100.000 | rlock = 1.483.654.721<br>wlock=100.000 | rlock =1.695.179.308<br>wlock=100.000 | rlock =1.686.820.809<br>wlock = 100.000 |

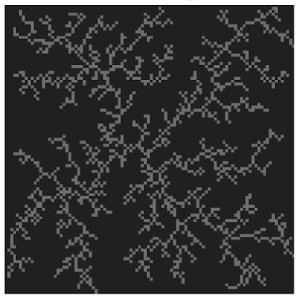
## Considerazioni finali

Nella prima implementazione, utilizzando  PThread e  Read/Write Lock, abbiamo ottenuto pessimi risultati per via dell'enorme numero di lock che venivano effettuati. Invece nell'ultima implementazione (sia PThread e OpenMP) attraverso l'uso di barriere abbiamo constatato che per problemi di piccola dimensione non ci sono miglioramenti, invece per problemi di grande dimensione i tempi sono dimezzati rispetto all'implementazione seriale.

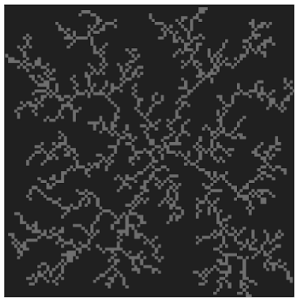
[1] https://en.wikipedia.org/wiki/Diffusion-limited_aggregation
[2] https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock

matrice 100x100 con 10.000 iterazioni e 1500 particelle

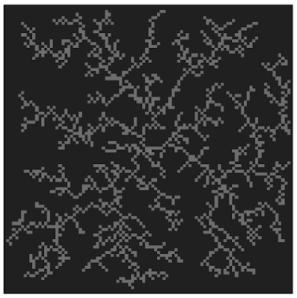
Omp



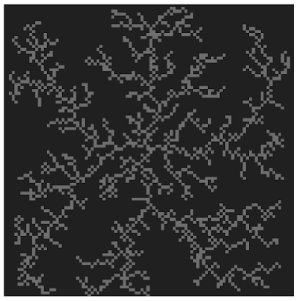
RWlock



Serial

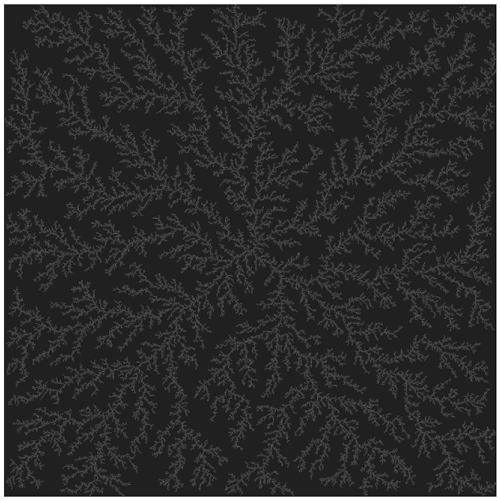


Barrier

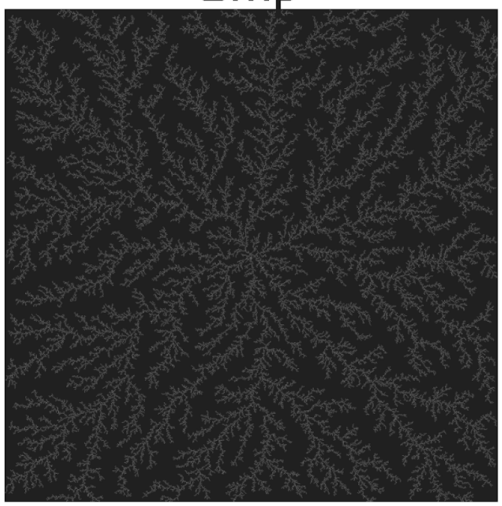


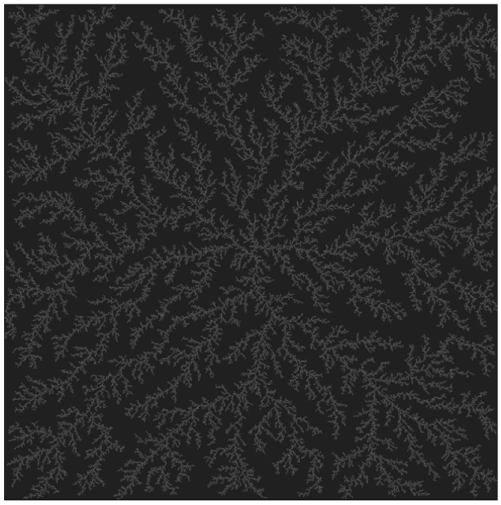
matrice 1000x1000 con 100.000 iterazioni e 100.000 particelle

Serial

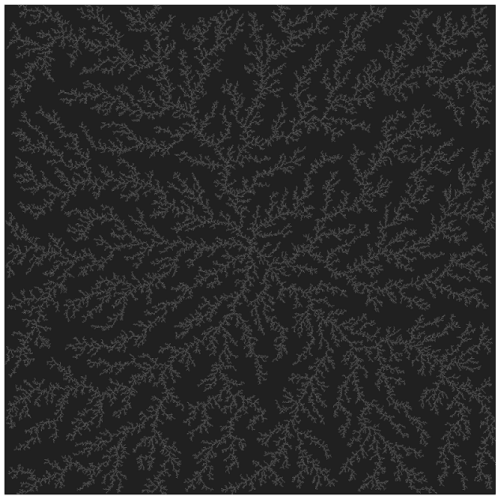


Omp





Barrier



RWlock