

PROGETTO CRESCITA CRISTALLINA CUDA & MPI

Tommaso Sgroi 1852992

Luca Masi 1969412

Crescita cristallina: è un fenomeno naturale in cui più particelle muovendosi nello spazio a disposizione, con moto browniano cioè in maniera casuale, si cristallizzano quando sono nelle vicinanze di un cristallo già formato. Per simulare tale fenomeno su di una macchina, abbiamo deciso di suddividere il tutto in due fasi:

1. movimento e pre-cristallizzazione
2. cristallizzazione

FASE 1

Le particelle controllano le celle adiacenti nella matrice per vedere se contengono un cristallo:

- In caso **positivo** allora entrano in precristallizzazione, quindi vengono rimosse o invalidate nella lista delle particelle e vengono aggiunte alla lista dei precristalli.
- In caso **negativo** allora la particella è libera di muoversi, sceglie quindi una direzione casuale verso cui muoversi quindi controlla se si trova nei limiti della matrice e decide se muoversi oppure rimanere nella posizione corrente

FASE 2

Tutte le particelle precedentemente salvate nella lista dei precristalli vengono cristallizzate ovvero vengono prese le coordinate e aggiornata la matrice.

La matrice è dove visualizzeremo il cristallo che si è formato una volta cristallizzate tutte le particelle. Tale matrice viene inizializzata con un solo cristallo ovvero il “seed” da cui poi si cristallizzeranno tutte le altre particelle.

Il seed è il cristallo madre da cui crescerà il cristallo finale.

STRUTTURE DATI

PARTICLE struttura di una particella, contiene:

- **x**: coordinata nella matrice.
- **y**: coordinata nella matrice.
- **rng**: seed per la generazione di numeri casuali (diverso da ogni altra particella).

SPACE struttura dello spazio in cui si muovono le particelle, contiene:

- **field****: matrice di interi in cui si muovono le particelle, una cella ha valore 1 se contiene un cristallo altrimenti 0.
- **len_x**: lunghezza dell'asse x.
- **len_y**: lunghezza dell'asse y.

ARRAYLIST lista dinamica che si aggiorna automaticamente all'inserire nuovi elementi, contiene:

- **particle***: array di particelle.
- **used**: lunghezza usata nella lista.
- **size**: lunghezza allocata reale della lista.

SINGLECORE

- Dichiaro e inizializzo gli arraylist delle particelle e delle particelle precristallizzate.
- Costruisco la matrice dei cristalli.

- Costruisco l'arraylist contenente le particelle.
- Muovo le particelle e in caso le precristallizzo.
- Cristallizzo le particelle dall'arraylist delle precristallizzate inserendo un 1 nella matrice.
- Ripeto i 2 passaggi precedenti finché o cristallizzo tutte le particelle o termina il numero di iterazioni.

MPI

Idea algoritmo:

Ogni host alloca le strutture dati nella sua memoria privata e le inizializza, viene creata una nuova struttura dati MPI_Datatype per le particelle. L'host 0 crea il vettore delle particelle e lo distribuisce verso tutti gli altri host.

Ogni host muove e precristallizza le sue particelle del vettore, poi ogni host raccoglie le particelle precristallizzate da tutti gli altri host, così ognuno cristallizza privatamente nella propria matrice.

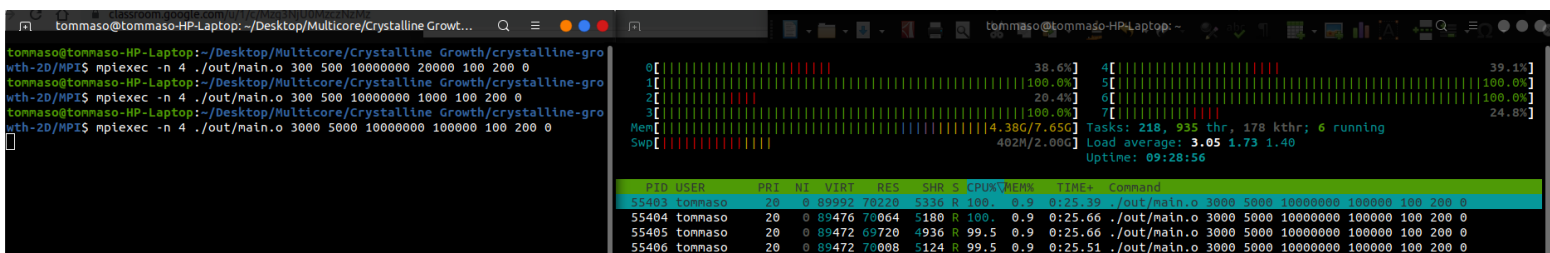
Architettura:

- Vengono dichiarate e inizializzate le strutture dati, in particolare viene costruita la MPI_Datatype per le particelle.
- L'host 0 costruisce il vettore delle particelle.
- Vengono calcolati di displacement per fare la scatternv e distribuire il vettore delle particelle tra gli host (l'host 0 prenderà anche le particelle in eccesso).
- Itera finché ogni processo o non ha più particelle o viene superato il numero di iterazioni:

- Muove e precristallizza le sue particelle locali e salva le particelle che hanno trovato un cristallo vicino nell'arraylist delle particelle precristallizzate.
- Applica una Allreduce per scambiare e ricevere la somma delle particelle rimanenti tra tutti gli host (serve per capire se gli host possono uscire dal ciclo).
- Usa una Allreduce sul numero delle particelle precristallizzate per sapere esattamente quanto iterare sul vettore.
- Usa una Allgather in modo che ogni host raccolga i numeri di particelle che si sono precristallizzate (serve per avere i displacement per fare successivamente la Allgatherv sulla lista dei precristalli).
- Calcola il displacement per effettuare la Allgatherv e raccogliere il numero esatto delle particelle precristallizzate.
- Effettua la Allgatherv scambiando esattamente il numero di particelle precristallizzate.
- Cristallizza le particelle.

Scelte progettuali:

Abbiamo deciso di allocare su tutti gli host la matrice dei cristalli e tenerla aggiornata tramite la comunicazione dei soli vettori delle particelle precristallizzate, in modo che tutti conoscano le posizioni dei cristalli. Ciò ci permette di evitare che un singolo host contenga la sola e unica matrice e tutti gli altri continuino a comunicargli tutti gli spostamenti a cui deve rispondere inoltrando le particelle cristallizzate. L'implementazione adottata ci permette di spartire in maniera equa il lavoro.



Abbiamo implementato gli arraylist che hanno permesso di mantenere ordinati gli array e tenere traccia degli elementi in maniera efficiente.

Correttezza:

Abbiamo testato mpi con vari valori e vari numeri di processi e abbiamo sempre riscontrato output identici a quelli dell'algoritmo singlecore, esempio:

```
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ ./main_single.o 5 10 10000 25 2 5 1
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C 0 0 C C C 0 C
0 C 0 C 0 C 0 C 0
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C 0 0 C C C 0 C
0 C 0 C 0 C 0 C 0
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ mpiexec -n 4 ./main_mpi.o 5 10 10000 25 2 5 1
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C 0 0 C C C 0 C
0 C 0 C 0 C 0 C 0
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ mpiexec -n 8 ./main_mpi.o 5 10 10000 25 2 5 1
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C 0 0 C C C 0 C
0 C 0 C 0 C 0 C 0
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ mpiexec -n 16 ./main_mpi.o 5 10 10000 25 2 5 1
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C 0 0 C C C 0 C
0 C 0 C 0 C 0 C 0
tommaso@tommaso-HP-Laptop:~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$
```

Input: matrice 5 x 10, Iterazioni 10000, particelle 25, posizione cristallo madre x = 2, y = 5, scrivi file = true (scrive su un file chiamato "output.space", dove 0 indica spazio vuoto e 'C' indica un cristallo).

Performance:

processi/input	M = 5 x 10 Iter. = 10000 Part. = 25 Seed = 2, 5	M = 20 x 30 Iter. = 10000 Part. = 500 Seed = 10, 15	M = 200 x 300 Iter. = 100000 Part. = 6000 Seed = 100, 150	M = 500 x 1000 Iter. = 100000 Part. = 300000 Seed = 500, 500
Single	0,003s	0,005s	1,920s	42,363s
2	0,045s	0,015s	1,312s	26,089s
3	0,034s	0,017s	0,950s	17,063s
4	0,033s	0,020s	1,093s	13,689s

Su piccoli input il singlecore è più performante in quanto il tempo di MPI per dividere le particelle e fare una continua comunicazione tra i processi è il principale bottleneck dell'intera esecuzione.

Infatti con un problema di dimensioni medio - basse (seconda colonna) possiamo già osservare che, riguardo MPI, meno processi sono presenti e meno comunicazioni avvengono diminuendo il tempo di esecuzione rispetto a problemi di dimensione bassa.

Invece un aumento delle performance si può osservare con problemi di dimensioni medio - alte e alte (terza e quarta colonna), dove all'aumentare del problema e dei processi si hanno dei netti e chiari miglioramenti, questo perché si possono muovere in parallelo sempre meno particelle per processo così che la comunicazione possa avvenire il prima possibile.

SPEEDUP:

processi/input	M = 5 x 10 Iter. = 10000 Part. = 25 Seed = 2, 5	M = 20 x 30 Iter. = 10000 Part. = 500 Seed = 10, 15	M = 200 x 300 Iter. = 100000 Part. = 5000 Seed = 100, 150	M = 500 x 1000 Iter. = 100000 Part. = 300000 Seed = 500, 500
2	0,067	0,33	1,46	1,62
3	0,088	0,29	2,02	2,48
4	0,090	0,25	1,75	3,09

Possiamo vedere come all'aumentare del problema aumenta anche lo speedup all'aumentare dei processi per input grandi.

EFFICIENZA:

processi/input	M = 5 x 10 Iter. = 10000 Part. = 25 Seed = 2, 5	M = 20 x 30 Iter. = 10000 Part. = 500 Seed = 10, 15	M = 200 x 300 Iter. = 100000 Part. = 5000 Seed = 100, 150	M = 500 x 1000 Iter. = 100000 Part. = 300000 Seed = 500, 500
2	0,0335	0,165	0,73	0,81
3	0,0293	0,096	0,673	0,826
4	0,0225	0,0625	0,4375	0,7725

Possiamo notare come all'aumentare del problema venga incrementa anche l'efficienza, che invece rimane bassa su input di grandezza medio - bassa.

CUDA

Idea algoritmo:

Si fa sempre un'associazione 1:1 tra thread e particelle quando si lancia un kernel.

uso 1 kernel per muovere e precristallizzare e 1 kernel per cristallizzare in modo che chiamando la `cudaDeviceSynchronize` posso aspettare la fine dell'esecuzione di tutti i blocchi prima di cristallizzare.

Il movimento e la precristallizzazione è stata pensata in modo da evitare la divergenza dei blocchi ed evitare il maggior numero di costose race conditions.

La cristallizzazione prende i dati dal vettore globale delle precristallizzate e aggiorna la matrice.

Per evitare di lanciare molti più thread del necessario abbiamo adottato una funzione per mantenere il vettore delle particelle contiguo.

Architettura:

- Riutilizzo le funzioni per costruire e inizializzare il campo degli altri algoritmi e converto la matrice in un vettore.
- Alloco lo spazio necessario sul device, si utilizzano anche delle `mallocManaged` in modo da facilitare la comunicazione di informazioni importanti sia per CPU che GPU come ad esempio il numero di particelle precristallizzate.
- Costruisco il vettore delle particelle lanciando un kernel.
- Itera finché o il numero di particelle rimanenti è maggiore di 0 oppure viene superato il numero di iterazioni:
 - Lancia un kernel per muovere e cristallizzare, il procedimento è molto simile ai precedenti algoritmi ma con la differenza che si dichiara una variabile `shared` per le particelle cristallizzate in un blocco che viene inizializzata solo dal thread 0 del blocco. Questa variabile serve a limitare l'accesso concorrente alla variabile globale del numero di

particelle cristallizzare globali, in quanto prima si accede concorrentemente alla variabile shared e infine il thread 0 del blocco aggiunge il numero di particelle cristallizzare shared al globale, così da avere un massimo di un accesso per blocco. Quando una particella viene cristallizzata viene invalidata sul vettore settando la sua x a -1.

- L'host attende la fine del kernel.
- Lancia un kernel per cristallizzare le particelle, in cui tutti settano la porzione della matrice indicata dalla particella a 1 e invalidano la particella settando la x a -1.
- L'host attende la fine del kernel.
- Per evitare di lanciare thread in eccesso si esegue un resize del vettore delle particelle rendendolo contiguo :
 - Copio il vettore delle particelle in un nuovo vettore, l'indice lo prendo accedendo sequenzialmente a una variabile globale tramite atomicAdd.
- L'host attende la fine del kernel.
- Per evitare di copiare sempre e tutte le particelle dal vettore che ha le particelle contigue e quello che contiene i buchi semplicemente scambio i puntatori dei vettori.
- Decrementa il numero di particelle del numero di particelle precristallizzate globalmente (in questo si lanceranno sempre meno blocchi).
- Resetta il numero di particelle precristallizzate globalmente a 0.

Scelte progettuali:

Abbiamo utilizzato il vettore dei precristalli di dimensioni pari a quello delle particelle che però all'esecuzione conterrà dei valori nulli, per i quali i thread che otterranno quelle particelle ritorneranno immediatamente dall'esecuzione. Questo è stato fatto perché non possiamo sapere dove, come e quando le particelle possono precristallizzarsi.

Abbiamo scelto di usare un'associazione 1:1 tra thread e particelle in modo da avere accessi efficienti in memoria.

Abbiamo scelto di mettere nella memoria shared, durante l'esecuzione del movimento e precristallizzazione, un intero per tenere il conto delle cristallizzate per blocco che poi verranno sommate da un solo thread sulla variabile globale delle particelle precristallizzate. Questo è stato fatto per

limitare sia l'interazione con la memoria globale che per evitare un'esecuzione sequenziale da parte di tutti i thread su un'unica variabile.

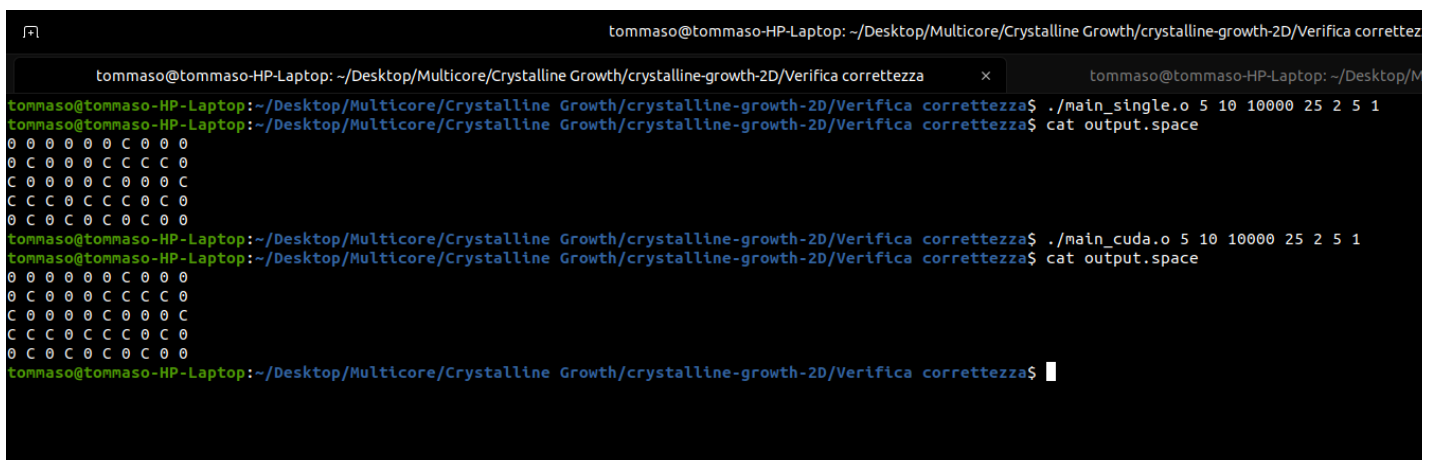
Limitazioni:

L'impossibilità di poter effettuare una sincronizzazione dei blocchi nel kernel, che è stata aggirata attendendo la fine dell'esecuzione del kernel tramite la `cudaDeviceSynchronize`. Questo metodo ha fatto incrementare i tempi di esecuzione a causa delle numerose chiamate `cudaLaunchKernel`.

L'utilizzo di un algoritmo di sorting per eliminare le particelle cristallizzate sul vettore delle particelle, diventava il principale bottleneck dell'intera esecuzione del programma. Mentre tenere i "buchi" (particelle invalide, quindi già cristallizzate) nel vettore delle particelle faceva sì che il lancio del kernel per il movimento diventasse il principale bottleneck.

Correttezza:

Abbiamo testato CUDA con vari valori e abbiamo sempre riscontrato output identici a quelli dell'algoritmo singlecore, esempio:



```
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ ./main_single.o 5 10 10000 25 2 5 1
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C C 0 C C C 0 C 0
0 C 0 C 0 C 0 C 0 0
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ ./main_cuda.o 5 10 10000 25 2 5 1
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$ cat output.space
0 0 0 0 0 0 C 0 0 0
0 C 0 0 0 C C C C 0
C 0 0 0 0 C 0 0 0 C
C C C 0 C C C 0 C 0
0 C 0 C 0 C 0 C 0 0
tommaso@tommaso-HP-Laptop: ~/Desktop/Multicore/Crystalline Growth/crystalline-growth-2D/Verifica correttezza$
```

Input: matrice 5 x 10, Iterazioni 10000, particelle 25, posizione cristallo madre $x = 2$, $y = 5$, scrivi file = true (scrive su un file chiamato "output.space", dove 0 indica spazio vuoto e 'C' indica un cristallo).

Performance:

thread/input	M = 5 x 10 Iter. = 10000 Part. = 25 Seed = 2, 5	M = 20 x 30 Iter. = 10000 Part. = 500 Seed = 10, 15	M = 200 x 300 Iter. = 100000 Part. = 6000 Seed = 100, 150	M = 500 x 1000 Iter. = 100000 Part. = 300000 Seed = 500, 500
Single	0,003s	0,005s	1,920s	42,363s
Pari al numero di particelle	0,093s	0,066s	2,132s	5,323s

SPEEDUP:

thread/input	M = 5 x 10 Iter. = 10000 Part. = 25 Seed = 2, 5	M = 20 x 30 Iter. = 10000 Part. = 500 Seed = 10, 15	M = 200 x 300 Iter. = 100000 Part. = 5000 Seed = 100, 150	M = 500 x 1000 Iter. = 100000 Part. = 300000 Seed = 500, 500
Pari al numero di particelle	0,0322	0,075	0,9	7,958

Possiamo osservare che hanno grossi miglioramenti più è grande l'input.