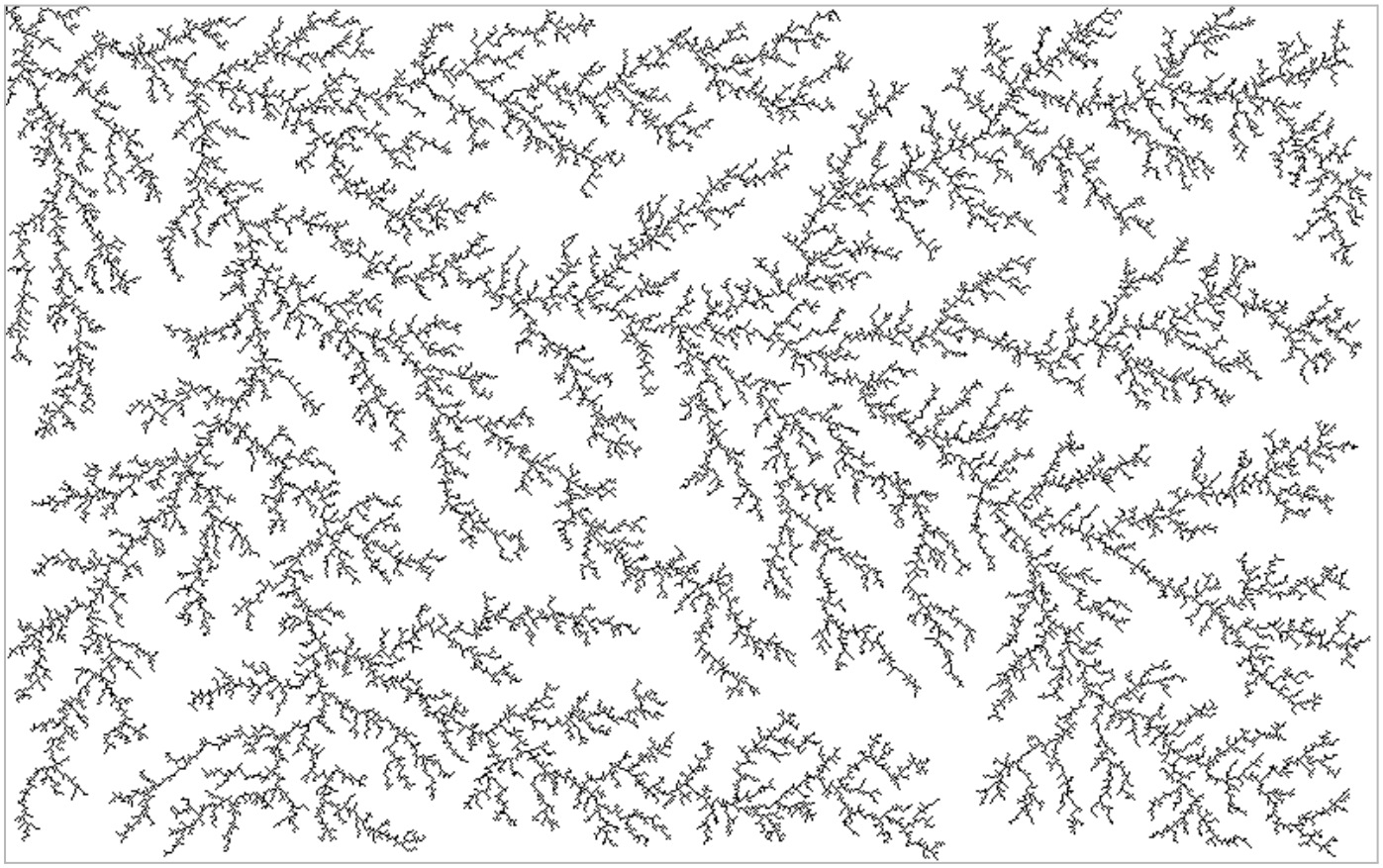
Progetto Programmazione di Sistemi Embedded e Multicore

Diffusion-Limited Aggregation

Daniele Venturini

1991255

02/2024



Indice

Introduzione

Diffusion-limited aggregation (DLA) è un processo di formazione di cristalli nel quale le particelle si muovono in uno spazio 2D con moto browniano (cioè in modo casuale) e si combinano tra loro quando si toccano. DLA può essere simulato utilizzando una griglia 2D in cui ogni cella può essere occupata da uno o più particelle in movimento.
Una particella diventa parte di un cristallo (e si ferma) quando si trova in prossimità di un cristallo già formato.

Implementazione

L'algoritmo utilizzato per simulare il DLA è stato implementato in modo seriale e parallelo, utilizzando le librerie OpenMP e MPI.
I parametri in input della simulazione sono:

- la dimensione della griglia (lunghezza e altezza);
- il numero di iterazioni;
- il numero di particelle;
- coordinate del cristallo iniziale (x e y);
- il numero di thread (solo con OpenMP).

Alla base l'algoritmo è rimasto lo stesso per tutte e tre le implementazione, di seguito i principali punti chiave:

1. Allocare dinamicamente la memoria per la griglia (tramite un semplice array a 2 dimensioni causava segmentation fault con input grandi);
2. Inizializzare la griglia con valori 255 (EMPTY) e il cristallo iniziale con 1 (CRYSTALIZED). Questo per avere sfondo bianco e cristalli neri nell'immagine finale;
3. Inizializzare le particelle con coordinate randomiche;
4. Inizio simulazione per ogni iterazione:
    4.1. Per ogni particella:
        4.1.1. Simula movimento browniano tra -1 e 1 in x e y;
        4.1.2. Controllo se rimane all'interno della griglia;
        4.1.3. Controllo nei dintorni della particella se esiste un cristallo;
        4.1.4. Cristallizza particella se vero.

Queste sono le parti del programma che vengono misurate per calcolare il tempo di esecuzione dell'algoritmo, il quale viene stampato in output in microsecondi.
Le particelle sono state implementate tramite una struttura con due interi per salvare le posizioni x e y nella griglia.
Alla conclusione dell'algoritmo, la griglia viene salvata in un'immagine di output nel formato ppm per consentire la visualizzazione dei risultati della simulazione.

Implementazione seriale

Inizialmente è stata scritta la versione seriale per capire il funzionamento dell'algoritmo e per fornire una base da cui trarre le versioni parallele dell'algoritmo.
Dopo aver ricevuto in input i parametri necessari, l'algoritmo alloca dinamicamente una griglia bidimensionale di interi di dimensioni height per righe e width per colonne e inizializza l'array di particelle con coordinate casuali.
Successivamente per ogni iterazione e per ogni particella, si calcola lo spostamento casuale e si controlla se ha nei dintorni un cristallo. Nel caso risulti vero, si cristallizza la particella e la si rimuove dall'array delle particelle (sostituendola con l'ultima particella dell'array e diminuendo il numero di particelle totali). In questo modo si velocizza il ciclo man mano che le particelle si cristallizzano.
Infine salva l'immagine in formato ppm e libera la memoria allocata dinamicamente.
Ovviamente questa implementazione risulta molto veloce con input di piccole dimensioni, andando a rallentare sempre di più con il crescere dell'input.

Implementazione con OpenMP

Con OpenMP sono state fornite due versioni. Di base l'algoritmo è lo stesso della versione seriale, tranne per le direttive di OpenMP per parallelizzare l'inizializzazione delle particelle e la simulazione del moto browniano durante le iterazioni.
Inizialmente nella prima versione, si era scelto di condividere l'array di particelle, assegnando una parte dell'array ad ogni thread, ed usare la direttiva "omp for" per parallelizzare il ciclo sulla simulazione del moto browniano.

Essendo il ciclo sul numero di particelle, questo comporta l'impossibilità di modificarne il numero come nell'algoritmo seriale a causa della limitazione della direttiva "omp for", quindi è stato aggiunto un controllo se la particella risulta già cristallizzata prima di simulare il moto.
In seguito si è scelto di non usare la direttiva, ma dividere tramite indici le parti di array su cui itera ogni thread, dato che risultava più veloce della controparte con la direttiva "omp for".
Nella seconda versione, l'array di particelle viene diviso per ogni thread, i quali si salvano su un proprio array le particelle su cui poi verrà simulato il moto browniano.
In questo modo risulta possibile ridurre il numero di particelle man mano che si cristallizzano.
Entrambe le versioni soffrono di una distribuzione ineguale del lavoro, ma questo problema è più evidente nella seconda versione. Tuttavia, la seconda versione risulta essere più veloce.
Sono state incluse entrambe le versioni per evidenziare la differenza nelle prestazioni dei due algoritmi, che differiscono solo leggermente.

Implementazione con MPI:

Con MPI sono state fornite tre versioni e l'algoritmo è stato un po' modificato per gestire in modo migliore la memoria condivisa.
In tutte e tre le versioni la griglia viene inizializzata solo dal primo processo in una finestra di MPI, ovvero in uno spazio di memoria accessibile dagli altri processi del programma, la griglia diventa un array lungo altezza per larghezza, per facilitare l'allocazione della memoria, e la cristallizzazione delle particelle avviene in un separato for su un array dedicato, dopo aver mosso tutte le particelle e salvato quelle da cristallizzare.
E' stata scelta questa soluzione per evitare di scambiare la griglia o gli aggiornamenti ad essa tra i processi durante l'esecuzione di ogni iterazione, ritenendo fosse più efficiente avere una sola griglia su cui i processi operano.
Nella prima versione il primo processo inizializza la griglia nella sua finestra e i vari processi mettono un solo lock di lettura sulla finestra prima di iniziare le iterazioni.
Per modificare la griglia, ovvero per cristallizzare una particella, i processi eseguono una MPI_Raccumulate con il parametro MPI_REPLACE perchè garantisce una operazione atomica, senza necessità di effettuare un lock in scrittura ogni volta che un processo deve cristallizzare delle particelle (Il che richiede un grande dispendio di risorse). Inoltre, MPI_Raccumulate si differenzia dalla MPI_Accumulate perché è una funzione non bloccante.
Nella seconda e terza versione, questa finestra viene allocata, sempre dal primo processo, in uno spazio di memoria unificato tra tutti i processi (Tramite MPI_Win_allocate_shared), garantendo l'accesso alla memoria tramite un semplice puntatore invece di usare funzioni RMA di MPI.
La seconda versione utilizza un lock esclusivo prima di iterare sulle particelle per modificare la griglia, togliendo il lock shared e rimettendolo dopo.
La terza versione utilizza una barriera prima di iterare sulle particelle da cristallizzare e una dopo, in modo da garantire due fasi distinte nell'algoritmo: una di sola lettura e una di sola scrittura.

Infatti anche se due processi modificano entrambi la stessa cella, il valore finale della cella sarà comunque il valore del cristallo.
Come in OpenMP, tutte le versioni soffrono di una distribuzione ineguale del lavoro a causa del diminuire delle dimensioni dell'array di particelle su cui opera ogni processo. La prima versione risulta essere decisamente troppo lenta, la seconda non porta a risultati più efficienti del seriale, mentre l'ultima risulta essere poco meno efficiente rispetto alla seconda versione di OpenMP.

Test cases

Input piccolo: 100x100, 10.000 iterazioni, 2.000 particelle, particella iniziale x=50 y=50

Input medio: 500x500, 50.000 iterazioni, 20.000 particelle, particella iniziale x=250 y=250

Input grande: 1000x1000, 100.000 iterazioni, 100.000 particelle, particella iniziale x=500 y=500

Tutti i test sono stati svolti su Windows in WSL con processore AMD Ryzen 5 3600 3.6GHz Base Clock e 16GB di RAM.

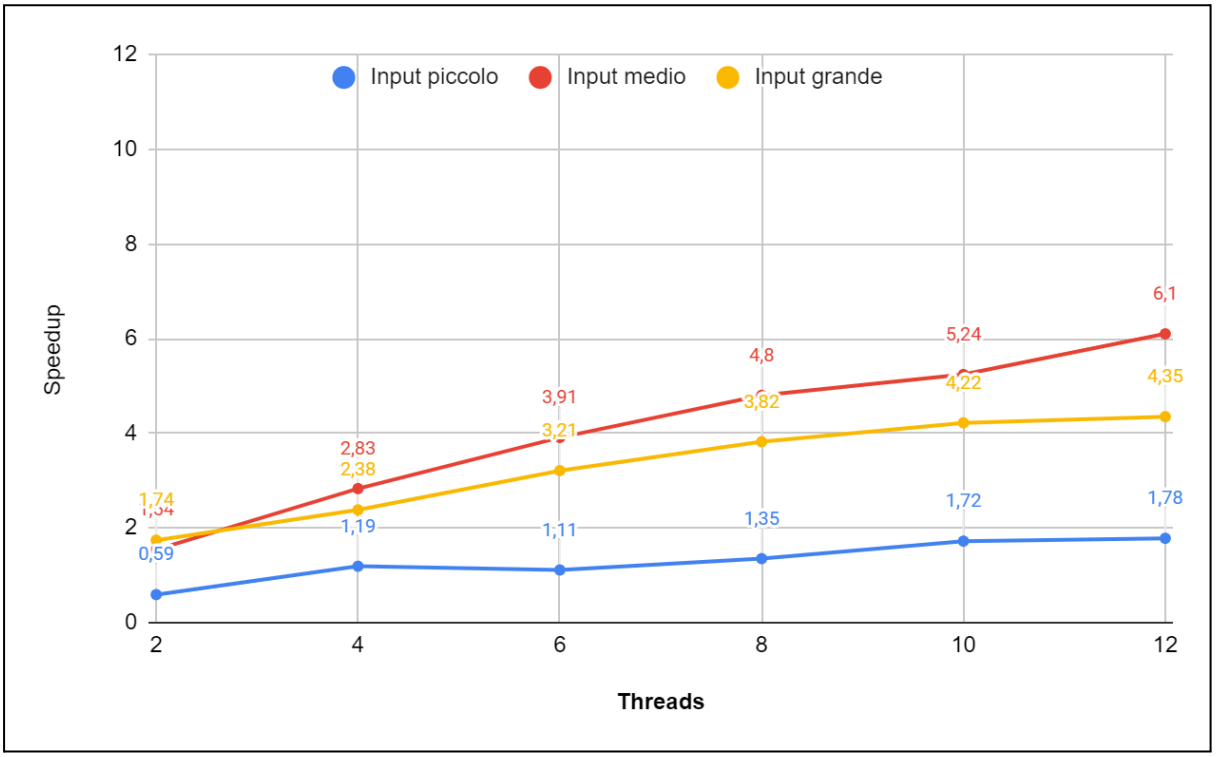
1 - Seriale

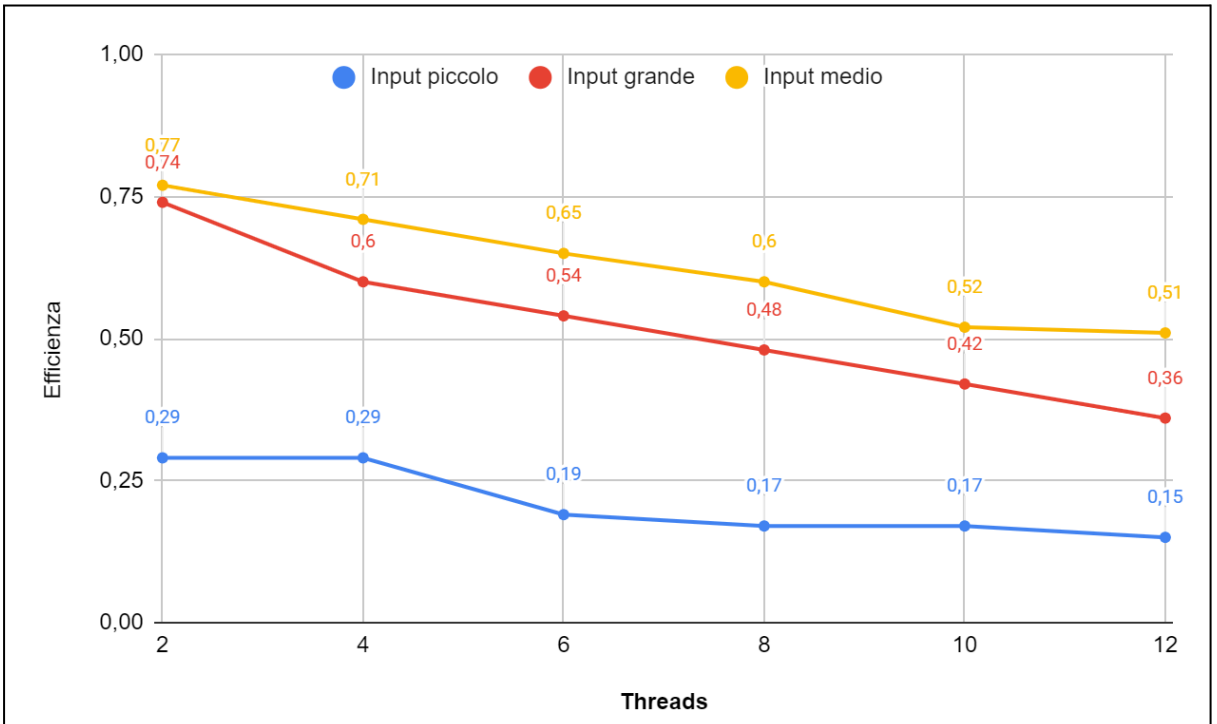
Input piccolo	Input medio	Input grande
21.480us	14.480.964us	88.413.931us

2 - openmp

Thread	Input piccolo	Input medio	Input grande
Seriale	21.480us	14.480.964us	88.413.931us
2 Speedup Efficienza	36.629us 0,59 0,29	9.388.664us 1,54 0,77	60.030.346us 1,47 0,74
4 Speedup Efficienza	18.099us 1,19 0,29	5.126.103us 2,83 0,71	37.179.039us 2,38 0,60
6 Speedup	19.306us 1,11	3.705.596us 3,91	27.569.675us 3,21

Efficienza	0,19	0,65	0,54
8 Speedup Efficienza	15.949us 1,35 0,17	3.022.707us 4,80 0,60	23.133.126us 3,82 0,42
10 Speedup Efficienza	12.512us 1,72 0,17	2.763.240us 5,24 0,52	20.971.397us 4,22 0,42
12 Speedup Efficienza	12.098us 1,78 0,15	2.377.632us 6,10 0,51	20.348.729us 4,35 0.36



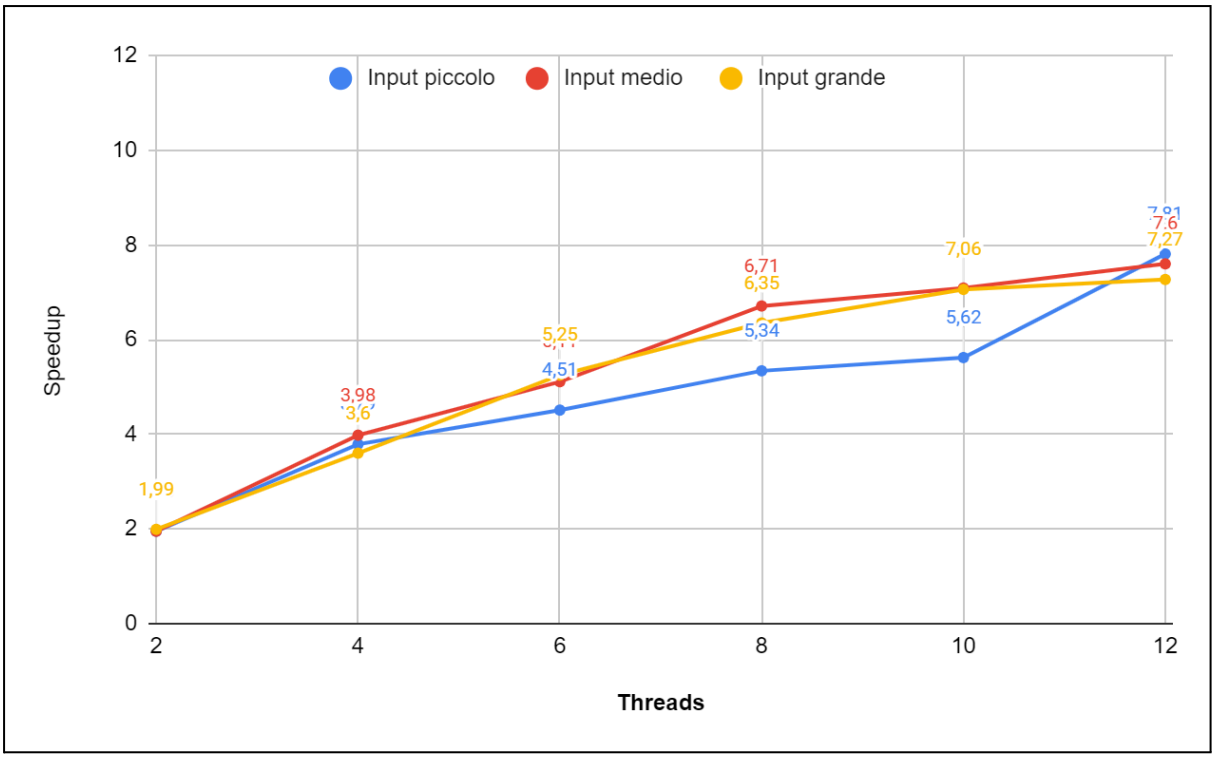


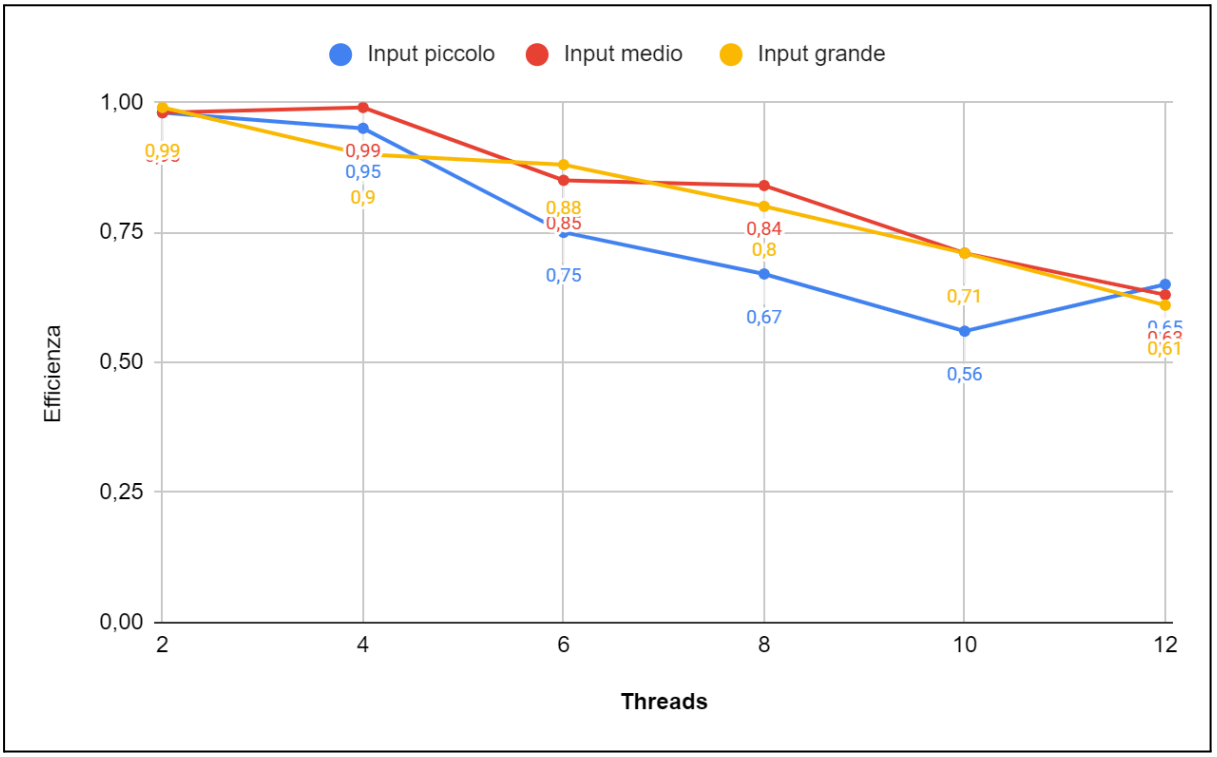
Considerazioni

Questa soluzione non offre un vantaggio con input di piccole dimensioni, andando invece a migliorare con input di dimensioni maggiori, ma non offre mai risultati eccellenti. La causa va a trovarsi nell'impossibilità di rimuovere le particelle una volta cristallizzate dall'array, andando a scorrere l'intero array ad ogni iterazione. Inoltre, come sarà in ogni soluzione proposta, la distribuzione ineguale del lavoro nel corso delle iterazioni è ciò che causa il calare continuo dell'efficienza dell'algoritmo.

3 - 2openmp

Thread	Input piccolo	Input medio	Input grande
Seriale	21.480us	14.480.964us	88.413.931us
2 Speedup Efficienza	11.030us 1,95 0,98	7.410.097us 1,95 0,98	44.260.632us 1,99 0,99
4 Speedup Efficienza	5.668us 3,79 0,95	3.636.182us 3,98 0,99	24.627.003us 3,60 0,90
6 Speedup Efficienza	4-760us 4,51 0,75	2.834.767us 5,11 0,85	16.844.785us 5,25 6,35
8 Speedup Efficienza	4.012us 5,34 0,67	2.155.722us 6,71 0,84	13.933.353us 6,35 0,80
10 Speedup Efficienza	3.830us 5,62 0,56	2.042.166us 7,09 0,71	12.526.034us 7,06 0,71
12 Speedup Efficienza	2.747us 7,81 0,65	1.906.274us 7,60 0,63	12.165.378us 7,27 0,61





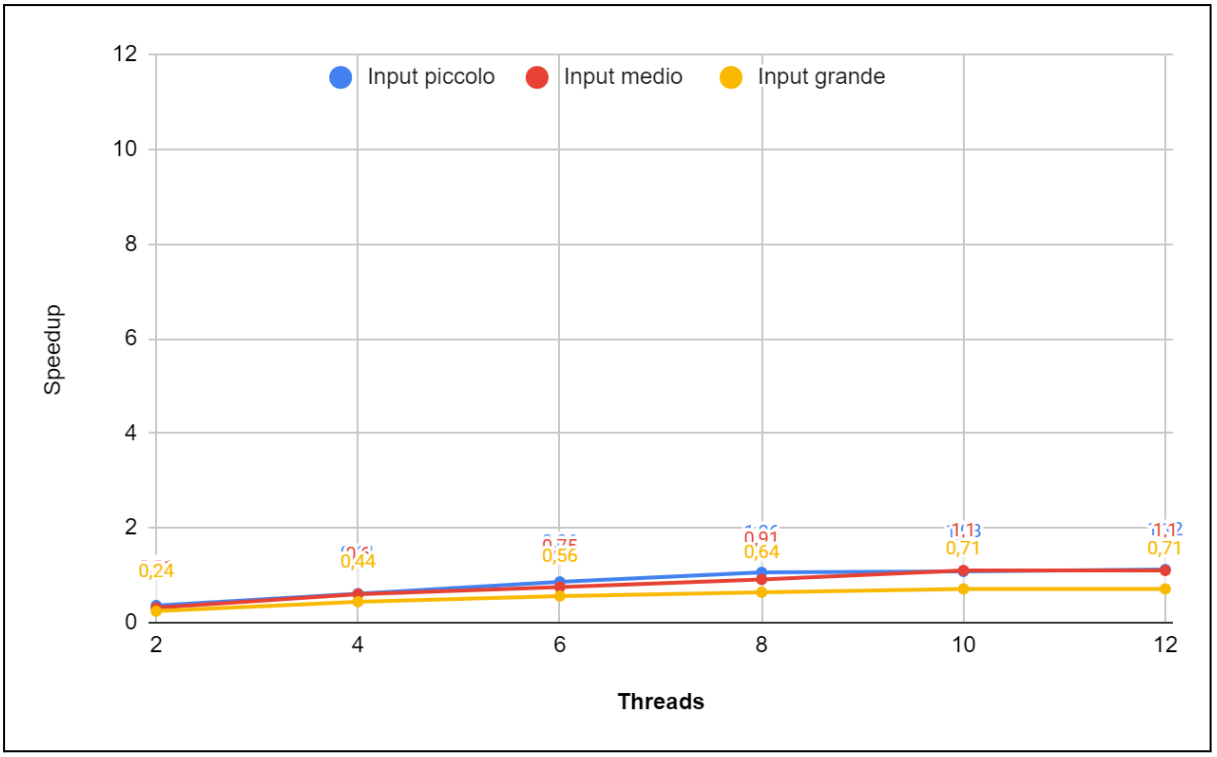
Considerazioni

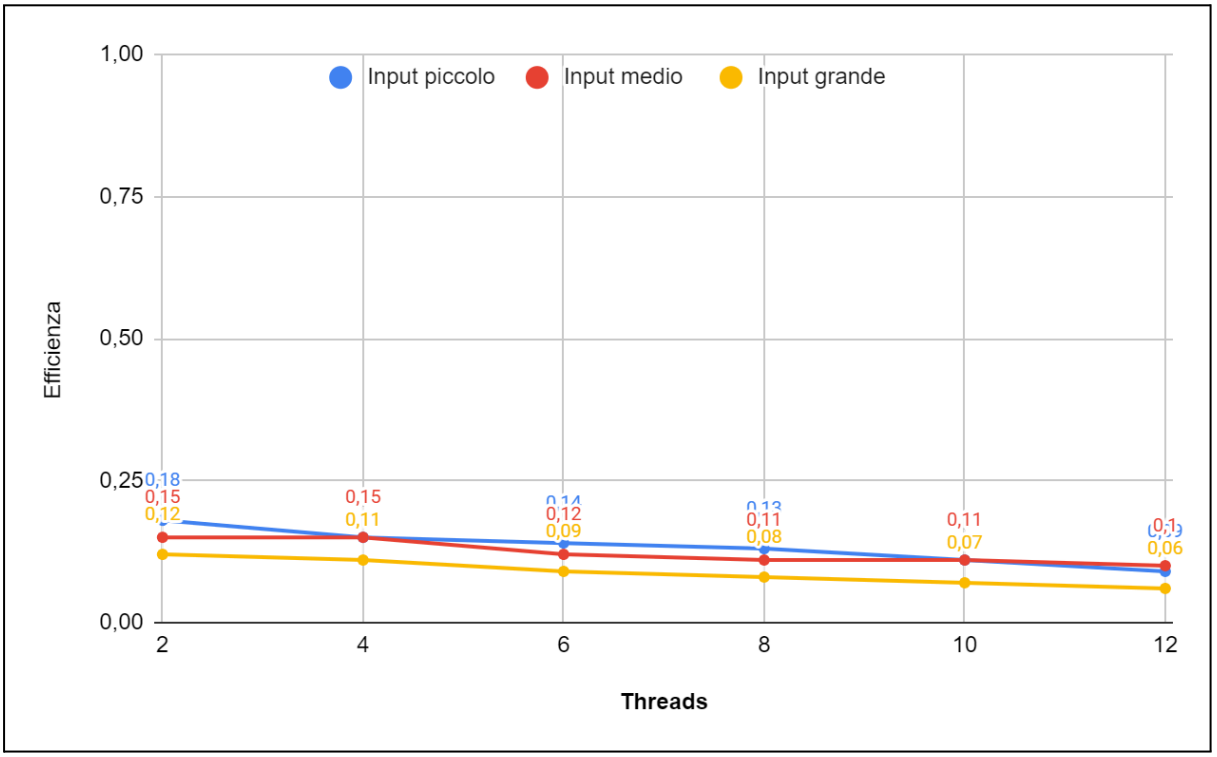
Questa soluzione offre i risultati migliori, anche con input di piccole dimensioni, e ottiene risultati quasi perfetti con pochi thread. Con molti thread l'efficienza va a diminuire, ottenendo comunque un gran speedup rispetto alla versione seriale, con la causa che dovrebbe essere sempre la distribuzione ineguale del lavoro tra i thread durante le iterazioni.

Da come si può notare con l'input piccolo, OpenMP sembra introdurre pochissimo overhead di parallelizzazione, portando a buoni risultati già da subito.

4 - mpi

Thread	Input piccolo	Input medio	Input grande
Seriale	21.480us	14.480.964us	88.413.931us
2 Speedup Efficienza	59.206us 0,36 0,18	47.306.126us 0,31 0,15	360.728.509us 0,24 0,12
4 Speedup Efficienza	35.444us 0,61 0,15	24.041.322us 0,60 0,15	203.217.868us 0,44 0,11
6 Speedup Efficienza	25.047us 0,86 0,14	19.409.665us 0,75 0,12	158.168.771us 0,56 0,09
8 Speedup Efficienza	20.247us 1,06 0,13	15.883.683us 0,91 0,11	138.549.306us 0,64 0,08
10 Speedup Efficienza	19.906us 1,08 0,11	13.214.108us 1,10 0,11	125.382.169us 0,71 0,07
12 Speedup Efficienza	19.207us 1,12 0,09	13.202.265us 1,10 0,10	123.914.951us 0,71 0,06


Input piccolo
Input medio
Input grande
Speedup
Threads
12
10
8
6
4
2
0
2
4
6
8
10
12
0,24
0,44
0,56
0,64
0,71
0,71

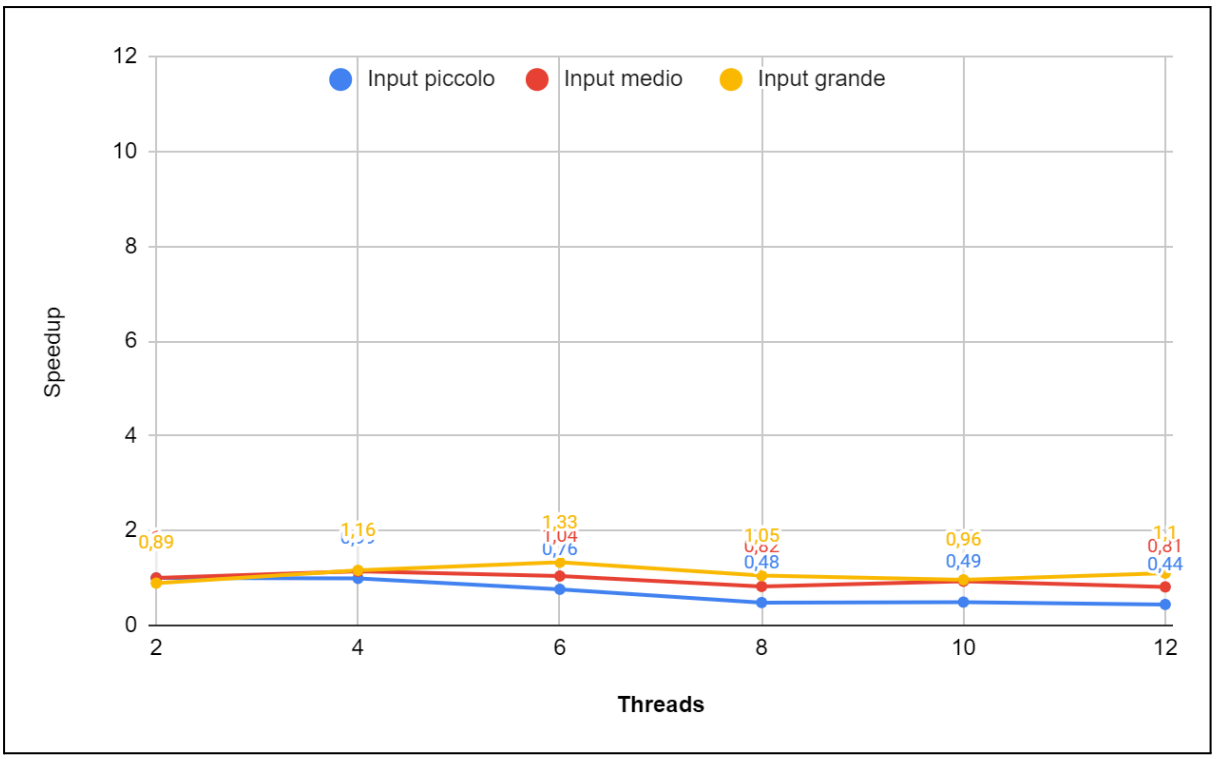

Input piccolo
Input medio
Input grande
Efficienza
Threads
1,00
0,75
0,50
0,25
0,00
2
4
6
8
10
12
0,18
0,15
0,12
0,15
0,11
0,14
0,12
0,09
0,13
0,11
0,08
0,11
0,07
0,06

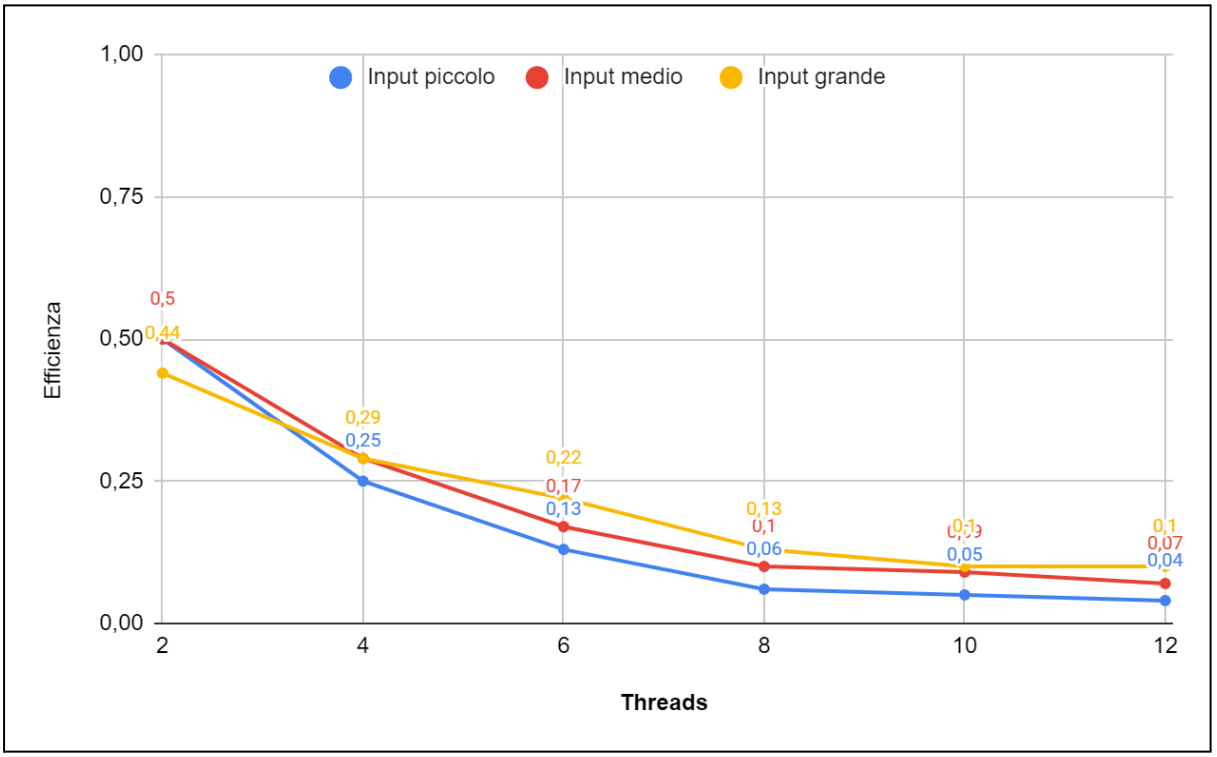
Considerazioni

Questa soluzione non riesce a migliorare in nessun caso rispetto all'algoritmo seriale, anzi andando anche a peggiorare in quasi ogni test con qualsiasi numero di thread.
La causa potrebbe essere il numero di chiamate alle funzioni RMA MPI_Raccumulate utilizzate per aggiornare la griglia nella memoria del processo 0 quando una particella si cristallizza, causando troppo overhead di comunicazione ad ogni aggiornamento. Anche le funzioni MPI_Put e MPI_Rput sono state testate, utilizzando lock esclusivi. Tuttavia, queste soluzioni hanno mostrato un'efficienza ancora peggiore rispetto a MPI_Raccumulate, principalmente a causa dell'utilizzo eccessivo di lock.
Questa soluzione è stata comunque inclusa nei test in quanto rappresenta la soluzione "migliore" trovata con l'utilizzo di una finestra allocata in memoria non unificata, diversamente dalle due soluzioni successive, al fine di evidenziare le differenze che possono sorgere utilizzandola.

5 - 2mpi

Thread	Input piccolo	Input medio	Input grande
Seriale	21.480us	14.480.964us	88.413.931us
2 Speedup Efficienza	21.485us 1,00 0,50	14.579.387us 1,00 0,50	99.558.905us 0,89 0,44
4 Speedup Efficienza	21.752us 0,99 0,25	12.764.091us 1,14 0,29	76.188.200us 1,16 0,29
6 Speedup Efficienza	28.447us 0,76 0,13	13.915.639us 1,04 0,17	66.336.992us 1,33 0,22
8 Speedup Efficienza	44.433us 0,48 0,06	17.724.513us 0,82 0,10	84.514.252us 1,05 0,13
10 Speedup Efficienza	44.022us 0,49 0,06	15.555.715us 0,93 0,09	92.568.670us 0,96 0,10
12 Speedup Efficienza	48.450us 0,44 0,04	17.970.068us 0,81 0,07	80.552.878us 1,10 0,10





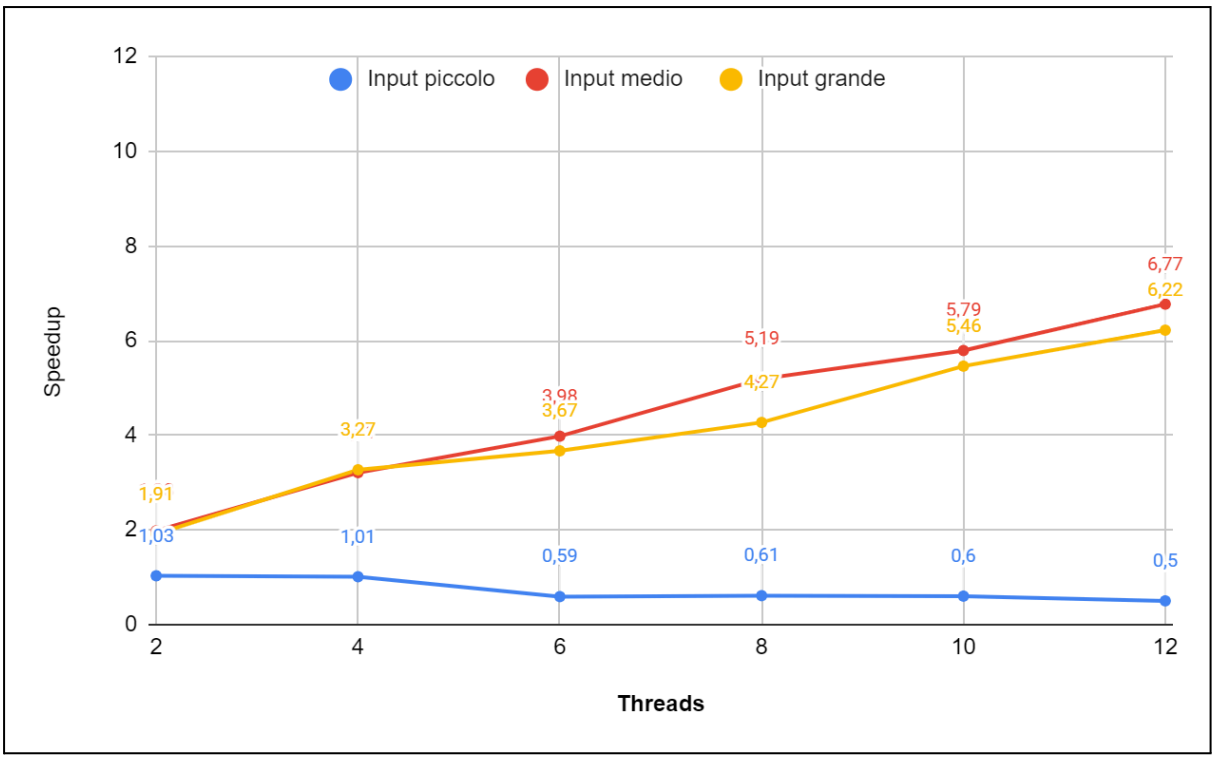
Considerazioni

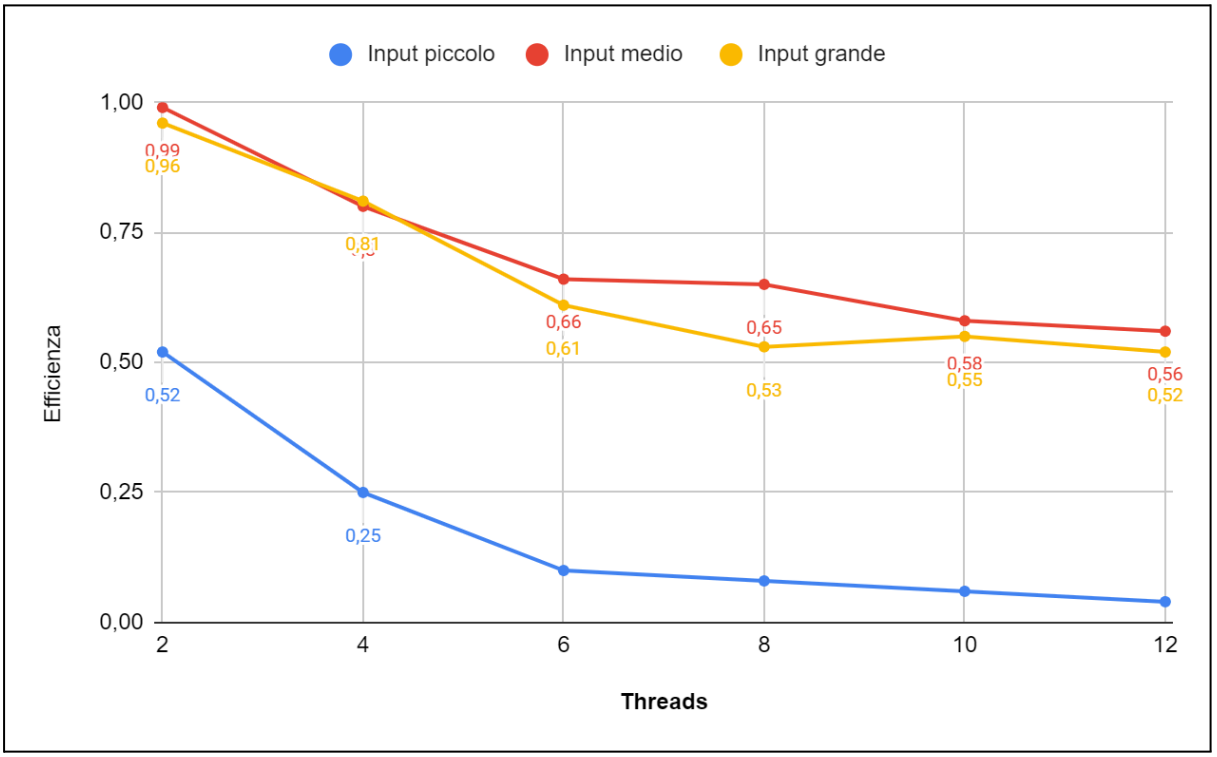
Questa soluzione mostra risultati leggermente migliori a partire dall'input di medie dimensioni, ma l'algoritmo rimane allineato con il tempo di esecuzione dell'algoritmo seriale. Infatti sia lo speedup che l'efficienza rimangono costantemente bassi e non migliorano con l'aumentare delle dimensioni dell'input. A differenza della soluzione precedente in MPI, non

si osserva un peggioramento del tempo di esecuzione rispetto all'algoritmo seriale, tuttavia l'eventuale miglioramento è ancora insufficiente.
Il motivo di questi risultati potrebbe derivare dall'eccessivo utilizzo di lock durante le iterazioni, nonostante l'utilizzo di una griglia memorizzata in una memoria unificata tra i processi.
Questa soluzione è stata inclusa nei test per evidenziare le differenze tra l'utilizzo di lock e l'utilizzo di barriere per l'aggiornamento della griglia, come invece avviene nella soluzione successiva, pur utilizzando entrambe la memoria unificata.

6 - 3mpi

Thread	Input piccolo	Input medio	Input grande
Seriale	21.480us	14.480.964us	88.413.931us
2 Speedup Efficienza	20.832us 1,03 0,52	7.306.928us 1,98 0,99	46.266.573us 1,91 0,96
4 Speedup Efficienza	21.318us 1,01 0,25	4.511.538us 3,21 0,80	27.081.118us 3,27 0,81
6 Speedup Efficienza	36.385us 0,59 0,10	3.639.064us 3,98 0,66	24.075.333us 3,67 0,61
8 Speedup Efficienza	35.079us 0,61 0,08	2.787.926us 5,19 0,65	20.718.759us 4,27 0,53
10 Speedup Efficienza	35.763us 0,60 0,06	2.501.449us 5,79 0,58	16.188.210us 5,46 0,55
12 Speedup Efficienza	42.819us 0,50 0,04	2.139.776us 6,77 0,56	14.222.517us 6,22 0,52





Considerazioni

Questa soluzione rappresenta il risultato migliore ottenuto utilizzando MPI.
I vantaggi non sono immediatamente evidenti con input di dimensioni ridotte, probabilmente a causa dell'overhead introdotto dalla parallelizzazione con MPI.

Tuttavia, già con input di dimensioni medie si registrano risultati ottimali con un numero limitato di thread, simili a quelli ottenuti con la soluzione 2openmp.
Purtroppo, anche in questa implementazione, l'efficacia diminuisce all'aumentare del numero di thread, principalmente a causa della distribuzione ineguale del carico di lavoro tra i processi, subendo un impatto ancor più significativo rispetto alla soluzione 2openmp.
Questo decremento delle prestazioni potrebbe essere attribuibile alla presenza di barriere utilizzate per sincronizzare le iterazioni tra i processi.