Automa Cellulare

L'obiettivo principale di questa applicazione in C++ è simulare un automa “Preda predatore” utilizzando la libreria grafica Allegro5 per avere un riscontro visivo ed ottimizzando le prestazioni attraverso un'implementazione parallela dell'algoritmo utilizzando MPI. Inoltre, viene impiegato anche POSIX per parallelizzare ulteriormente l’algoritmo. “Preda predatore”, così come ogni altro algoritmo che simula un automa cellulare è un sistema computazionale astratto che si è dimostrato molto utile sia come modello generale di complessità sia come rappresentazione più specifica di dinamiche non lineari in una varietà di campi scientifici.

**Regole**

Abbiamo 3 tipologie di celle nella nostra matrice:

**Spazio vuoto:**

Se ci sono almeno due prede vicine, c'è una piccola possibilità che diventi una preda.

Se ci sono abbastanza prede e almeno un predatore portiere, può diventare un predatore.

Le probabilità che uno spazio vuoto diventi una preda sono del 20%

Altrimenti resta vuoto.

**Preda:**

Se c'è almeno un predatore vicino, la preda viene mangiata (diventa spazio vuoto).

Altrimenti resta preda.

**Predatore:**

Se non ci sono prede o ci sono troppi predatori vicini (3), il predatore muore di fame (diventa spazio vuoto).

Altrimenti resta predatore.

Per “vicino” si intende adiacente, nelle otto caselle intorno alla cella che stiamo controllando.

**Struttura del Progetto**

Il progetto è strutturato come segue:

* serial\_project: Versione seriale del progetto con la libreria grafica Allegro.
* serial\_project\_without\_allegro: Versione seriale del progetto senza la libreria grafica Allegro.
* main\_project: Versione parallela del progetto che utilizza MPI+Posix.
* main\_project\_without\_allegro: Versione parallela del progetto che utilizza MPI+Posix, senza la libreria grafica Allegro.

**Cosa fa questo progetto?**

Il presente elaborato descrive l'implementazione di un automa cellulare per la simulazione dell'interazione tra prede e predatori in un ambiente bidimensionale. Vengono presentate quattro versioni del programma: una seriale e una parallela basata su MPI + Posix, ed altre 2 senza un’interfaccia grafica.

**Implementazione:**

Il progetto si basa su un'architettura ibrida che sfrutta sia la memoria distribuita (MPI) che quella condivisa (thread Posix) per massimizzare le prestazioni e la scalabilità. Il sistema è in grado di eseguire un automa cellulare complesso in modo parallelo, suddividendo il dominio dell'automa su entrambi gli assi (X e Y) e distribuendo i calcoli tra più processi MPI e thread Posix.

**Funzionalità:**

La piattaforma sfrutta il parallelismo sia a livello di processi (MPI) che a livello di thread (Posix) per distribuire il carico computazionale in modo uniforme, sfruttando al meglio le risorse hardware disponibili. Il progetto può gestire matrici di grandi dimensioni garantendo prestazioni elevate e scalabilità.

Un visualizzatore basato sulla libreria Allegro permette di osservare l'evoluzione dell'automa in tempo reale, facilitando l'analisi del comportamento del sistema.

Le impostazioni di configurazione possono essere cambiate dall’apposito file *config.cfg* oppure verranno usati i valori de default. Nel progetto, ogni processo MPI gestisce una porzione orizzontale della matrice di simulazione, scambiando i bordi superiori e inferiori con i processi adiacenti tramite comunicazioni punto a punto (**MPI\_Isend, MPI\_Irecv**). Per ottimizzare le prestazioni, la comunicazione tra processi avviene tramite scambio di messaggi (**MPI\_Gather**) e si impiega l'invio asincrono di messaggi con MPI\_Isend. Per quanto riguarda l’algoritmo di controllo delle regole invece, vengono usati dei thread Posix che dividono la matrice gestita da MPI in verticale. Visto che questi thread hanno una memoria condivisa, non necessitano di uno scambia bordi (destro o sinistro) a differenza di MPI. Dopo l’esecuzione degli steps indicati nel file cfg, il programma termina, mostrando i risultati.

**Funzionalità aggiuntive:**

Il programma include la generazione di numeri random thread-safe e la gestione della terminazione controllata tramite il tasto "ESC".

Il programma al termine degli steps, scrive su file i tempi dell’elaborazione ed altri valori.

Viene sfruttata un’implementazione di un parser json per cambiare le impostazioni di configurazione a piacimento, in ogni run.

**Test**

I test sono stati eseguiti su una macchina con CPUAMD Ryzen 5 4600H @ 3.000GHz con architettura 6+6 core e 20GB di RAM.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Matrice 1000\*1000 (100 steps)** | | | | | |
| **Tempo seriale** | **Tempo parallelo** | **Numero thread mpi** | **Speedup** | **Efficienza** | **Overhead** |
| 6.01 | 3.12 | 2 | 1.92 | 0.96 | 0.24 |
|  | 2.13 | 4 | 2.82 | 0.70 | 2.52 |
|  | 1.62 | 6 | 3.70 | 0.62 | 3.72 |
|  | 1.37 | 8 | 4.38 | 0.55 | 4.96 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Matrice 2000\*2000 (100 steps)** | | | | | |
| **Tempo seriale** | **Tempo parallelo** | **Numero thread mpi** | **Speedup** | **Efficienza** | **Overhead** |
| 24.03 | 12.14 | 2 | 1.98 | 0.99 | 0.25 |
|  | 6.4 | 4 | 3.75 | 0.94 | 1.57 |
|  | 4.63 | 6 | 5.19 | 0.87 | 3.75 |
|  | 5.49 | 8 | 4.38 | 0.55 | 19.89 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Matrice 3000\*3000 (100 steps)** | | | | | |
| **Tempo seriale** | **Tempo parallelo** | **Numero thread mpi** | **Speedup** | **Efficienza** | **Overhead** |
| 56.35 | 28.17 | 2 | 2 | 1 | 0.01 |
|  | 14.75 | 4 | 3.82 | 0.96 | 2.65 |
|  | 10.42 | 6 | 5.41 | 0.9 | 6.17 |
|  | 11.26 | 8 | 5 | 0.63 | 33.73 |





