Relazione Progetto Algoritmi Paralleli e Sistemi Distribuiti

Introduzione

Il progetto rappresenta la versione parallela dell'automa cellulare "Il Gioco della Vita".

Il Gioco della Vita è un automa cellulare, nel quale ogni cellula può appartenere a due stati differenti, *VIVA* o *MORTA*, seguendo delle semplici regole:

1. Una cellula viva continua a vivere se ha esattamente due o tre cellule vicine vive;
2. Una cellula morta torna in vita se ha esattamente tre cellule vicine vive;
3. Una cellula viva muore per spopolamento se ha un numero di cellule vicine vive minore o uguale ad uno;
4. Una cellula viva muore per sovrappopolamento se ha un numero di cellule vicine vive maggiore o uguale a quattro.

Seguendo queste semplici regole, ogni nuova generazione dell'automa cellulare, dipendente solo dalla precedente generazione, diventa, appunto, il nuovo modello sulla quale si basa la simulazione della generazione successiva.

Rappresentazione dell'automa

L'automa viene rappresentato tramite un array dinamico monodimensionale di tipo booleano, dove ogni cella può essere:

- *false*, che rappresenta una cellula *MORTA*
- *true*, che rappresenta una cellula *VIVA*

La scelta di utilizzare array monodimensionali ricade sul fatto che in questo modo le celle sono memorizzate in modo lineare in memoria. Questo concetto è fondamentale per poter utilizzare con facilità le funzioni della libreria *MPI*.

Due versioni

Sono stati prodotti due codici, il primo (*grafica.cpp)*, ovvero il più completo, prevede una parte grafica realizzata con la libreria *Allegro5*, mentre la seconda versione (*tempi.cpp)*, è identica alla prima, con la differenza che sono state rimosse tutte le funzioni che si occupano della grafica e sono state aggiunte un paio di variabili con lo scopo di fornire in output i tempi, lasciando però l'algoritmo intatto.

L'algoritmo e l'implementazione

Il progetto è stato scritto utilizzando il linguaggio *C++* e la libreria *MPI* che ha permesso la parallelizzazione dell'algoritmo. L'uso di un ambiente di sviluppo basato sul *message passing* prevede l'uso di funzioni di *send* e *receive* tra i vari processi. L'architettura è del tipo *Master/Slave*, quindi prevede che un processo *Master* si occupa della distribuzione dei dati, della ricezione dei dati una volta lavorati dagli *Slaves* e della grafica, mentre i processi *Slave* (incluso il *Master*) ricevono i dati, li lavorano e successivamente vengono rispediti al mittente.

Nel progetto, solo il processo *Master* ha la matrice principale, che viene inizializzata. Ad ogni generazione il master invia a tutti gli altri processi singole porzioni di matrice, attraverso la funzione *MPI_Scatter*, in modo che ogni processo modifichi la sua matrice

locale e poi la rispedisca al processo *Master*, attraverso la funzione *MPI_Gather*, che si occuperà poi della stampa.

Lavorare semplicemente la propria sottomatrice locale non basta però! Si deve gestire accuratamente il caso in cui un processo abbia bisogno, solo in lettura, delle celle adiacenti ad una determinata cella, che però sono possedute da un altro processo. Se non venisse gestito questo caso, ogni processo lavorerebbe solo con la sua sottomatrice locale e il risultato sarebbe che la simulazione non sarebbe per niente corretta!

Nel progetto, il problema è stato gestito grazie all'uso di array di supporto che memorizzano, per ogni processo, l'orlo esterno superiore e quello inferiore, ovvero l'ultima riga del processo precedente e la prima riga del processo successivo, quando esistono. Questo ci permette di accedere alle celle adiacenti a quelle che si trovano sul bordo superiore e su quello inferiore, risolvendo il problema. Ad ogni generazione, quindi, ogni processo invia la propria riga superiore al processo precedente e la propria riga inferiore al processo successivo (se il processo è il primo, quindi non ha nessun processo precedente, non invierà la propria riga superiore, stesso discorso per l'ultimo processo che non invierà la propria riga inferiore). Questo meccanismo però provoca sicuramente un *deadlock* tra i vari processi, perché tutti inviano e si bloccano, aspettando la ricezione degli altri, che non avverrà mai! È possibile risolvere questo ulteriore problema grazie ad un trucchetto: i processi con *idProc* pari faranno prima le due *MPI_Send* e poi le due *MPI_Receive*, invece i processi con *idProc* dispari faranno prima le due *MPI_Reicive* e poi le due *MPI_Send*. Questo ci permette di evitare il *deadlock* tra i processi e risolve definitivamente il problema di gestione delle cosiddette *Ghost Cells*.

Ogni processo ora si occupa di aggiornare la propria sottomatrice, usando una sottomatrice di supporto per preservare la copia originale di ogni generazione. La matrice aggiornata viene mandata indietro al processo *Master* che si occuperà della stampa.

I risultati

TEMPI				
PROCESSORI	N = 500	N = 1000	N = 2500	N = 5000
1	0,72749	2,85819	18,30033	74,82740
2	0,38303	1,53694	9,83314	39,34711
3	0,26694	1,06490	6,80936	27,35604
4	0,20717	0,83790	5,24106	20,94845

SPEEDUP ($S = t_s / t_p$)				
PROCESSORI	N = 500	N = 1000	N = 2500	N = 5000
1	1,0	1,0	1,0	1,0
2	1,9	1,9	1,9	1,9
3	2,7	2,7	2,7	2,7
4	3,5	3,4	3,5	3,6

EFFICIENZA ($E = S / p$)				
PROCESSORI	N = 500	N = 1000	N = 2500	N = 5000
1	1,00	1,00	1,00	1,00
2	0,95	0,93	0,93	0,95
3	0,91	0,89	0,90	0,91
4	0,88	0,85	0,87	0,89

Nelle tre tabelle precedenti è possibile osservare come, all'aumentare delle dimensioni del problema (con *N* il numero di righe e di colonne della matrice principale, quindi composta da *N* * *N* celle), e all'aumentare del numero di processori, lo *Speedup* è ottimo, come lo è anche l'*efficienza*. Questo indica che il programma (*tempi.cpp)* scala molto bene ed è molto efficiente.

Istruzioni per la compilazione ed esecuzione

Per compilare il file *grafica.cpp (*è necessario che siano installate le librerie *Allegro* e *MPI):*

```
mpiCC grafica.cpp -o grafica -lallegro -lallegro_primitives -lallegro_font -lallegro_ttf
```

Per compilare il file *tempi.cpp (*è necessario che sia installata la libreria *MP):I*

```
mpiCC tempi.cpp -o tempi
```

Per eseguire il file *grafica.cpp (*sostituire [NUMERO_PROCESSI] con un numero):

```
mpirun -np [NUMERO_PROCESSI] ./grafica
```

Per eseguire il file *tempi.cpp (*sostituire [NUMERO_PROCESSI] con un numero):

```
mpirun -np [NUMERO_PROCESSI] ./tempi
```