PROGETTO ALGORITMI PARALLELI E SISTEMI DISTRIBUITI

Game of Life

Vecchio Francesco - 219901

OBIETTIVO

Il progetto prevede lo sviluppo di un sistema che emula un modello basato sugli automi cellulari operando in parallelo. È implementato utilizzando un approccio ibrido che integra sia la memoria condivisa, tramite l'utilizzo di POSIX Threads (Pthreads), sia la memoria distribuita attraverso il Message Passing Interface (MPI).

Al centro del sistema vi è la `transitionFunction`, la quale riceve coordinate `x` e `y` per applicare le regole del gioco della vita. La configurazione iniziale del gioco è fornita da un file denominato `input.txt`, il quale detiene lo stato iniziale della griglia di gioco.

Le specifiche operative, quali il numero di partizioni orizzontali (asse x) e verticali (asse y) della griglia, il numero di thread per processo MPI e il totale degli step evolutivi da eseguire, sono definite all'interno di un secondo file, `configuration.txt`. Questo permette una facile adattabilità e configurazione del progetto in base a esigenze di scalabilità e prestazioni differenti.

```
void initAllPartitions() {

    // Inizializza le dimensioni delle partizioni
    nRowsPerPartition = new int[nPartY]; //quante righe di celle per ogni partizione
    nColsPerPartition = new int[nPartX]; //quante colonne di celle per ogni partizione

    // Trova un divisore comune (i) per nThreads e nPartX o nThreads e nPartY
    //(in base alla formula size * nThreads = nPartX * nPartY si sa che ne esiste sempre uno)
    // Decide COME suddividere la matrice totale tra i processi (quali sottomatrici assegnare a ciascun
    for (int i = nThreads; i > 0; i--) {
        if ((nThreads % i == 0) && (nPartX % i == 0)) { //se i è un divisore comune di nThreads e nPart
            nPartYPerProc = nThreads / i; //per fare in modo che ad ogni thread venga assegnata una par
            nPartXPerProc = i;
            break;
        } else if ((nThreads % i == 0) && (nPartY % i == 0)) { //se i è un divisore comune di nThreads
            nPartYPerProc = i;
            nPartXPerProc = nThreads / i;
            break;
        }
    }

    //Distribuisco le righe e le colonne tra le partizioni (considerando anche il resto, ovvero il rem
    distributeRowsAndCols(totRows, nPartY, nRowsPerPartition);
    distributeRowsAndCols(totCols, nPartX, nColsPerPartition);

    // Calcola il numero di processi lungo l'asse X
    // Quante sottomatrici finiscono su X
    nProcOnX = nPartX / nPartXPerProc;

    // Scomponi il rank del processo ⟶ COORDINATE DEL PROCESSO
    rankX = rank % nProcOnX;
    rankY = rank / nProcOnX;

    // Calcola le dimensioni della sottomatrice per il processo corrente
    nRowsThisRank = sumPartitions(rankY * nPartYPerProc, nPartYPerProc, nRowsPerPartition);
    nColsThisRank = sumPartitions(rankX * nPartXPerProc, nPartXPerProc, nColsPerPartition);

    // Inizializza matrici di lettura e scrittura
    readM = new int[(nRowsThisRank + 2) * (nColsThisRank + 2)]; //+2 perchè si considerano le halo cell
    writeM = new int[(nRowsThisRank + 2) * (nColsThisRank + 2)];
}
```

PARTIZIONAMENTO

La funzione `initAllPartitions()` è progettata per impostare la griglia di partizioni. Stabilisce due array, `nRowsPerPartition` e `nColsPerPartition`, che definiscono la dimensione di ogni partizione lungo gli assi X e Y.

Per assicurarsi che ogni thread possa lavorare efficacemente, senza lasciare alcuna partizione senza risorse di calcolo, la funzione trova un divisore comune tra il numero totale di thread e il numero di partizioni orizzontali o verticali.

Successivamente determina come allocare le sottomatrici tra i processi. Lo scopo è quello di garantire un carico di lavoro bilanciato e minimizzare il sovraccarico comunicativo tra i processi.

Infine, la funzione distribuisce le righe e le colonne tra le partizioni in modo dinamico, tenendo conto anche del resto della divisione.

```c
void * runThread(void * arg){
    int tid = *(int *)arg;
    int tidX = tid % nPartXPerProc;  //calcolo l'tid lungo lasse X della partizione
    int tidY = tid / nPartXPerProc;
    int partXPrevious = rankX * nPartXPerProc + tidX;  //numero di partizioni lungo l'asse X che ci sono prima dell'attuale
    int partYPrevious = rankY * nPartYPerProc + tidY;
    int beginThreadX = 1; //la partizione inizia dalla prima riga. (perchè la 0 è halo)
    int beginThreadY = 1;

    for(int j = rankX * nPartXPerProc; j<partXPrevious; ++j) beginThreadX += nColsPerPartition[j];
    for(int i = rankY * nPartYPerProc; i<partYPrevious; ++i) beginThreadY += nRowsPerPartition[i];
    for(int s = 0; s < step; s++){
        comunicationBarrier();   //sincronizzo tutti i thread prima di procedere. Mi devo assicurare che tutti i processi abbiano ricevuto i dati corretti prima di continuare
        //Perche serve la barrier di comunicazione? mentre i thread vengono creai ed eseguono la loro funzione, il mainThread è andao avanti nel main è sta facendo
        // la exchangeBorders
        execTransFunc(beginThreadX, beginThreadY, nColsPerPartition[partXPrevious], nRowsPerPartition[partYPrevious]); //inizioX e Y, size X e Y
        completedBarrier(); //assicura che tutti i thread abbiano completato l'esecuzione di execTransFunc prima di procedere.Altrimenti alcuni potrebbero andare
        //al passo successivo mentre altri stanno ancora computando, possibili problemi.
    }
    return NULL;
}
```

BARRIER

La funzione `runThread()` applica le regole del Game of Life su ciascuna partizione. Per garantire l'integrità dei dati tra i passaggi di calcolo, vengono utilizzate delle `barrier` per sincronizzare i thread.

Le barrier assicurano che tutti i thread raggiungano un punto di sincronizzazione comune prima di procedere, prevenendo così conflitti di dati e garantendo uno scambio di informazioni coerente tra processi. Questo è particolarmente critico quando si scambiano i bordi delle partizioni, poiché un'informazione non aggiornata può portare a risultati non corretti.

La `execTransFunc()` è chiamata all'interno di un ciclo iterativo per ogni step di simulazione. Incapsula la logica per l'applicazione delle regole di transizione, ed è eseguita tra due barrier per assicurare che tutti i thread abbiano finito il loro lavoro prima di procedere al passo successivo.

La stessa logica di barrier viene utilizzata per coordinare il main thread con i thread di esecuzione, assicurando così un flusso di lavoro armonico e sincronizzato su tutto il sistema di calcolo distribuito.

```
158 void createDatatype(){
159     MPI_Type_vector(nRowsThisRank+2, 1, nColsThisRank+2, MPI_INT, &typeColumn);
160     MPI_Type_vector(nRowsThisRank, nColsThisRank, nColsThisRank+2, MPI_INT, &typeMatWithoutHalos);
161     MPI_Type_commit(&typeColumn);
162     MPI_Type_commit(&typeMatWithoutHalos);
163 }
```

DATATYPE

All'interno della funzione `createDatatype()`, è stato introdotto un datatype MPI personalizzato denominato `typeColumn`. Questo tipo di dato è stato specificamente progettato per facilitare lo scambio di colonne di dati, inclusi gli elementi *"halo"* che sono fondamentali per le operazioni nei bordi delle partizioni.

`typeColumn` è configurato per rappresentare una colonna singola di celle estendendosi per il numero di righe assegnate a ciascun processo (`nRowsThisRank`), più due righe aggiuntive per le *"halo cells"*. L'uso dello stride, che equivale al numero totale di colonne più due, permette di saltare efficacemente da una riga all'altra nella matrice globale.

Parallelamente, definiamo un secondo datatype, `typeMatWithoutHalos`, che facilita il trasferimento di dati dai **worker** al **main thread**, escludendo le *"halo cells"*. Questo assicura che la matrice aggregata per la visualizzazione sia priva di dati ausiliari, rendendo la rappresentazione grafica più accurata e riducendo il sovraccarico comunicativo.

L'adozione di questi datatypes personalizzati in MPI non solo semplifica le operazioni di invio e ricezione, ma ottimizza anche l'uso della banda e la chiarezza del codice. Ciò contribuisce a un significativo guadagno in termini di efficienza, specialmente in calcoli ad alta intensità comunicativa.

SCAMBIO BORDI

```
6  void sendRecvRows(int rankUp, int rankDown, MPI_Request &req) {
7      MPI_Isend(&readM[v(1,1)], nColsThisRank, MPI_INT, rankUp, 17, MPI_COMM_WORLD, &req
8      MPI_Isend(&readM[v(nRowsThisRank, 1)], nColsThisRank, MPI_INT, rankDown, 20, MPI_C
9
10     MPI_Recv(&readM[v(nRowsThisRank+1, 1)], nColsThisRank, MPI_INT, rankDown, 17, MPI_
11     MPI_Recv(&readM[v(0,1)], nColsThisRank, MPI_INT, rankUp, 20, MPI_COMM_WORLD, MPI_S
12 }
13 void sendRecvCols(int rankLeft, int rankRight, MPI_Request &req) {
14     MPI_Isend(&readM[v(0, 1)], 1, typeColumn, rankLeft, 12, MPI_COMM_WORLD, &req);
15     MPI_Isend(&readM[v(0, nColsThisRank)], 1, typeColumn, rankRight, 30, MPI_COMM_WORL
16
17     MPI_Recv(&readM[v(0, 0)], 1, typeColumn, rankLeft, 30, MPI_COMM_WORLD, MPI_STATUS_
18     MPI_Recv(&readM[v(0, nColsThisRank+1)], 1, typeColumn, rankRight, 12, MPI_COMM_WOR
19 }
20 void sendRecvCorners(int rankUpLeft, int rankUpRight, int rankDownLeft, int rankDownRi
21     MPI_Isend(&readM[v(1, 1)], 1, MPI_INT, rankUpLeft, 40, MPI_COMM_WORLD, &req);
22     MPI_Isend(&readM[v(1, nColsThisRank)], 1, MPI_INT, rankUpRight, 41, MPI_COMM_WORLD
23     MPI_Isend(&readM[v(nRowsThisRank, 1)], 1, MPI_INT, rankDownLeft, 42, MPI_COMM_WORL
24     MPI_Isend(&readM[v(nRowsThisRank, nColsThisRank)], 1, MPI_INT, rankDownRight, 43,
25
26     MPI_Recv(&readM[v(nRowsThisRank+1, 0)], 1, MPI_INT, rankDownLeft, 40, MPI_COMM_WOR
27     MPI_Recv(&readM[v(nRowsThisRank+1, nColsThisRank+1)], 1, MPI_INT, rankDownRight, 4
28     MPI_Recv(&readM[v(0, 0)], 1, MPI_INT, rankUpLeft, 42, MPI_COMM_WORLD, MPI_STATUS_I
29     MPI_Recv(&readM[v(0, nColsThisRank+1)], 1, MPI_INT, rankUpRight, 43, MPI_COMM_WORL
30 }
31 void exchangeBordersMore() {
32     MPI_Request req;
33     // Invio e ricezione delle righe superiore e inferiore
34     sendRecvRows(rankUp, rankDown, req);
35     // Invio e ricezione delle colonne sinistra e destra
36     sendRecvCols(rankLeft, rankRight, req);
37     // Invio e ricezione dei corner: in alto a sinistra, in alto a destra, in basso a
38     sendRecvCorners(rankUpLeft, rankUpRight, rankDownLeft, rankDownRight, req);
39 }
```

L'uso di funzioni MPI **Isend** per lo scambio non bloccante dei bordi permette ai processi di continuare le loro computazioni mentre i messaggi vengono inviati, ottimizzando il flusso di esecuzione e riducendo i tempi di attesa.

Le **Isend** giocano un ruolo fondamentale nel prevenire deadlock nella comunicazione. Lavorando in maniera asincrona, consentono ai thread di inviare dati senza rimanere in attesa di una corrispondente **Recv**.

Ogni processo invia e riceve i bordi simultaneamente, assicurando che ogni porzione della griglia sia aggiornata correttamente prima del successivo step.

Le funzioni `` `sendRecvRows()` ``, `` `sendRecvCols()` ``, e `` `sendRecvCorners()` `` sono implementate per gestire lo scambio di bordi orizzontali, verticali e degli angoli, rispettivamente. L'uso accurato dei tag MPI assicura che ogni messaggio venga correttamente identificato e gestito dal processo ricevente.

VALUTAZIONE DELLE PRESTAZIONI

Dim AC	N Step	Tempo Seriale (sec)	Tempo Parallelo (sec)	Speedup (Ts / Tp)	Efficienza (S / Np)
100	20	0.008240	0.007731	1.06	13.25%
300	20	0.089036	0.023587	3.77	47.12%
600	20	0.339724	0.103029	3.29	41.12%
1200	20	1.314189	0.348567	3.72	46.5%
5000	20	25.213600	5.352838	4.7	58.75%
10000	20	115.106815	24.150653	4.76	59.5%

Panoramica

Sono stati condotti test di performance per valutare la velocità e l'efficienza del programma parallelo. Attraverso l'uso di uno script Python, sono state generate matrici di dimensioni variabili al fine di testare il sistema in diversi scenari operativi.

Metodologia

- Dimensioni matrice (Dim AC): da 100 a 10000.
- Numero di step: 20 per tutti i test.
- Ripetizioni test: ogni configurazione è stata testata 10 volte per ottenere una media affidabile.
- Risorse di calcolo: 8 processori per l'esecuzione parallela.

VALUTAZIONE DELLE PRESTAZIONI

Dim AC	Speedup (Ts / Tp)	Efficienza (S / Np)
100	1.06	13.25%
300	3.77	47.12%
600	3.29	41.12%
1200	3.72	46.5%
5000	4.7	58.75%
10000	4.76	59.5%

Scalabilità e Speedup

- Dimostrato un aumento dell'efficacia con l'aumentare delle dimensioni della matrice.
- **Speedup** massimo di 4.76 per la matrice più ampia, dimostrando un notevole vantaggio del parallelismo.

Efficienza del Sistema

- Efficienza variabile tra il 41% e il 59%, con migliori risultati per problemi di dimensioni maggiori.
- Evidenziata l'importanza di ottimizzare la comunicazione tra i processori per sostenere l'efficienza.

Insights

- Per matrici piccole, il parallelismo risulta meno vantaggioso a causa dell'overhead di comunicazione.
- La legge di Amdahl suggerisce che l'overhead di gestione limita il miglioramento delle prestazioni per problemi di dimensioni minori.