Forest-Fire-Simulation

Introduzione

Il Forest-Fire-Model è un modello di cellular automata definito attraverso una matrice NxN. Ogni cella della matrice utilizzata in questo modello può assumere i seguenti stati:

- **TREE**
- **EMPTY**
- **BURNING_TREE**

All'interno del progetto rappresentati attraverso:

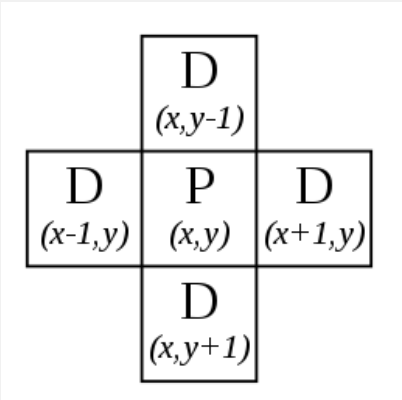
ASCII Charachter	Stato	Emoji
1	TREE	🌲
2	EMPTY	🚫
3	BURNING_TREE	🔥

Il modello è definito da 4 regole fondamentali:

1. Una cella in fiamme(**BURNING_TREE**) si trasforma in una cella vuota(**EMPTY**)
2. Un albero(**TREE**) andrà in fiamme se almeno un vicino è in fiamme(**BURNING_TREE**).
3. Un albero(**TREE**) andrà in fiamme con probabilità ***f*** anche se non ci sono vicini in fiamme(**BURNING_TREE**).
4. In una cella vuota(**EMPTY**) crescerà un albero(**TREE**) con probabilità ***p***.

Il concetto di vicinanza utilizzato

> **Quartiere di Von Neumann**
>
> Negli automi cellulari, il quartiere di von Neumann (o 4 quartieri) è classicamente definito su un bidimensionale reticolo quadrato ed è composto da una cella centrale e dalle sue quattro celle adiacenti. Il quartiere prende il nome John von Neumann, che lo ha utilizzato per definire il von Neumann automa cellulare.



Configurazione Ambiente ed Esecuzione

Prerequisiti

- Ubuntu 18.04 LTS
- Docker Per configurare l'ambiente di sviluppo è stato utilizzato un container Docker creato a partire da:

```
docker run -it --mount src="$(pwd)",target=/home,type=bind
spagnuolocarmine/docker-mpi:latest
```

L'ambiente effettivo di esecuzione ha comportato invece la creazione di un Cluster omogeneo formato da N macchine.
É stato utilizzato GCP(Google Cloud Platform) per la creazione del cluster composto da 6 macchine **e2-standard-4(4 vCPU, 16GB di Memoria)**.

La configurazione del cluster è stata realizzata mediante la seguente guida
https://github.com/spagnuolocarmine/ubuntu-openmpi-openmp

Soluzione Proposta

L'algoritmo è stato implementato attraverso il Linguaggio C ed OpenMPI, un'implementazione dello standard MPI(Message Passing Interface)

L'algoritmo prende in input N ed I, rispettivamente:

- N - Dimensione della Matrice NxN
- I - Numero di Iterazioni dell'algoritmo sulla Foresta

Il processo master si occupa della generazione di una matrice NxN che rappresenta la nostra foresta, viene poì calcolato il lavoro che spetta ad ogni processo slave e gli viene inviata la porzione di matrice da analizzare.

Successivamente ogni processo invia in maniera asincrona la propria porzione da analizzare ad i vicini e riceverà quindi dagli altri processi la loro parte.
Ogni processo(slave) durante la fase di comunicazione asincrona inizia ad analizzare la porzione della matrice assegnatagli, indipendente dai vicini, una volta ricevuti gli elementi dai vicini vengono applicate le regole che comportano il controllo dei vicini e successivamente, terminate le iterazioni vengono inviate al master le invia al master la porzione aggiornata.

Analisi del Codice

Analizziamo il codice associato alla generazione della foresta.

```
    //main_2.c

    char *forest = malloc(sizeof *forest * N * N); //Starter Forest Matrix
    char *temp = malloc(sizeof *forest * N * N); //Temp Matrix

    if(my_rank == 0){
        srand(42);//Random Seed
```

```
        generation(N,&forest);
    }
```

```
//myforest.h

#define F 10 //100    //Ignite probability F(Fire)
#define P 70 //100       //New Tree probability

#define TREE "🌲"
#define BURNING_TREE "🔥"
#define EMPTY "🚫"

// 1(TREE) 2(EMPTY) 3(BURNING TREE)

void generation(int N, char **matrix){
    srand(42);
    for(int i=0; i < N; i++){
        for(int j =0; j < N; j++){
            int randValue = (rand() % 101);
            if(randValue <= F){
                (*matrix)[(i*N)+j] = '3';
            }else if(randValue > F && randValue <= P){
                (*matrix)[(i*N)+j] = '1';
            }else{
                (*matrix)[(i*N)+j] = '2';
            }
        }
    }
}
```

In seguito occorre dividere il lavoro tra i processi slave ed inviargli le porzioni della foresta su cui lavorare.

```
    //main_2.c

    //Calcolo il numero di elementi per ogni processo
    int* sendCount = malloc ( size_p* sizeof(int));
    int* displacement = malloc(size_p * sizeof(int));
```

```
    //main_2.c
    divWork2(N,size_p,&sendCount,&displacement);
    char *recvBuff;
    if(my_rank == 1 || my_rank == (size_p-1)){
        recvBuff = (char*) calloc(sendCount[my_rank]+N,sizeof(char));
    }else{
        recvBuff = (char*) calloc(sendCount[my_rank]+(2*N),sizeof(char));
    }
```

```
    if(my_rank == 1){

MPI_Scatterv(forest,sendCount,displacement,MPI_CHAR,recvBuff,sendCount[my_rank],MP
I_CHAR,0,MPI_COMM_WORLD);
    }else{
        MPI_Scatterv(forest,sendCount,displacement,MPI_CHAR,
(recvBuff+N),sendCount[my_rank],MPI_CHAR,0,MPI_COMM_WORLD);
    }
```

```
//myforest.h

void divWork2(int N, int size, int** sendCount, int** displacement){
    int numberOfRow = N / (size - 1);
    int restVal = N % (size - 1);

    int pos = 0;
    (*displacement)[0] = 0;
    (*sendCount)[0] = 0; //Al processo master non dò nulla da fare.

    for(int i = 1; i < size;i++){
        (*sendCount)[i] = (numberOfRow*N);
        if(restVal > 0){
            (*sendCount)[i] += N;
            restVal--;
        }
        //Calcolo il displacement per la scatterv
        (*displacement)[i] = pos;
        pos = pos + (*sendCount)[i];
        //printf("Displ[%d]:%d \n",i,(*displacement)[i]);
        //printf("SendCount[%d]: %d \n ",i,(*sendCount)[i]);
    }
}
```

Ogni processo slave andrà ad inviare la propria porzione di matrice ad i vicini ed a riceverla dagli altri processi

```
        //main_2.c

        if( my_rank != 0){
            if(prec != -10){
                //Ricevo dal precedente
                MPI_Irecv(recvBuff,N,MPI_CHAR,prec,TAG,MPI_COMM_WORLD,&req1);
            }
            //Ricevo dal successivo
            if(dest != -10){
                if(my_rank == 1){

MPI_Irecv((recvBuff+sendCount[my_rank]),N,MPI_CHAR,dest,TAG,MPI_COMM_WORLD,&req2);
                }else{
```

```
MPI_Irecv((recvBuff+N+sendCount[my_rank]),N,MPI_CHAR,dest,TAG,MPI_COMM_WORLD,&req2
);
                }
            }
            //Invio al precedente
            if(prec != -10){
                MPI_Isend((recvBuff+N),N,MPI_CHAR,prec,TAG,MPI_COMM_WORLD,&req);
            }
            if(dest != -10){
                //Invio al successivo
                if(my_rank == 1){
                    MPI_Isend((recvBuff+sendCount[my_rank])-
N,N,MPI_CHAR,dest,TAG,MPI_COMM_WORLD,&req);
                }else{
                    MPI_Isend((recvBuff+N+sendCount[my_rank])-
N,N,MPI_CHAR,dest,TAG,MPI_COMM_WORLD,&req);
                }
            }
```

Si inizia quindi ad analizzare la porzione della matrice che già detiene il processo mentre si attendono gli elementi dei vicini

```
    //main_2.c

    //Lavoro sui miei elementi
    int start = my_rank == 1 ? 0 : 1;
    int end = my_rank == 1 ? sendCount[my_rank] : sendCount[my_rank] + N;

    checkMine(recvBuff,temp,start,end,my_rank,prec,dest, N); //flag a 0 check
senza vicini

    MPI_Wait(&req1,&Stat1);
    MPI_Wait(&req2,&Stat2);
```

```
//myforest.h
void checkMine(char* recvBuff, char* temp, int start, int end, int rank, int prec,
int dest,int N){
    for(int i = start; i < end/N; i++){
        for(int j = 0; j < N; j++){
                if(recvBuff[(i*N)+j] == '2'){ // 4) An empty space fills with a
tree with probability p
                    if((rand() % 101) <= P ){
                        temp[(i*N)+j] = '1';
                    }else{
                        temp[(i*N)+j] = '2';
                    }
                }else if(recvBuff[(i*N)+j] == '3') { //1) A burning cell turns
```

```
into an empty cell
                    temp[(i*N)+j] = '2';
                }
            }
        }
    }
```

Successivamente occorre analizzare i vicini degli elementi per determinare l'espandersi delle fiamme

```
            for(int i = start; i < end/N; i++){
                for(int j = 0; j < N; j++){
                    if(recvBuff[(i*N)+j] == '1'){
                        burningTree(temp,recvBuff,start,end,i, j, N);
                        //2) A tree will burn if at least one neighbor is burning
                        //3) A tree ignites with probability f even if no neighbor
is burning
                    }
                }
            }
```

Il tutto a partire dall'invio/ricezione dei vicini, all'interno di un ciclo che itera su I, infine viene effettuato uno swap dei puntatori per continuare a lavorare nelle successive iterazioni alla porzione di matrice aggiornata

```
        char* suppPointer;
        suppPointer = recvBuff;
        recvBuff = temp;
        temp = suppPointer;
```

Una volta terminate le iterazioni si procede con l'invio al processo master delle porzioni di matrice aggiornate da ogni processo slave

```
    if(my_rank == 1){

MPI_Gatherv(recvBuff,sendCount[my_rank],MPI_CHAR,forest,sendCount,displacement,MPI
_CHAR,0,MPI_COMM_WORLD);
    }else{

MPI_Gatherv(recvBuff+N,sendCount[my_rank],MPI_CHAR,forest,sendCount,displacement,M
PI_CHAR,0,MPI_COMM_WORLD);
    }
```

Correttezza della soluzione

Benchmark

Di seguito sono mostrate le misurazioni effettuate utilizzando il Cluster omogeneo già descritto .

Sono state effettuate le misurazioni per ottenere:

- **Scalabilità Forte**
- **Scalabilità Debole**

Scalabilità Forte

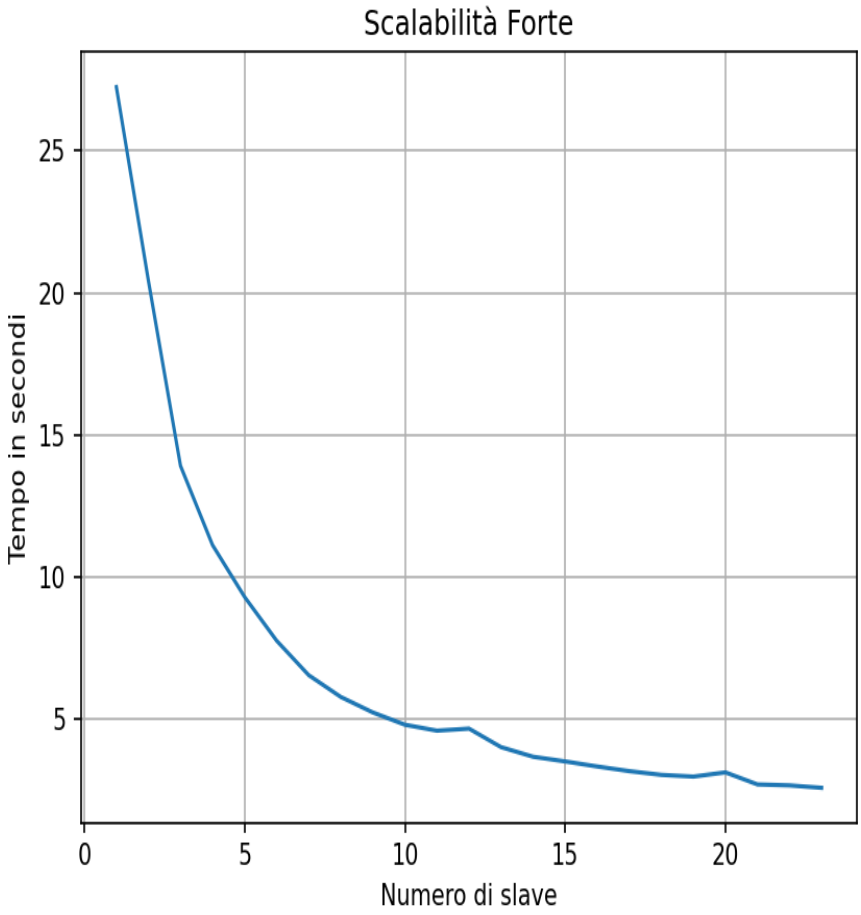
La scalabilità forte è stata ottenuta analizzando più esecuzioni dell'algoritmo, utilizzando il Cluster configurato, aumentando il numero di processi ad ogni esecuzione, utilizzando quindi da 2 vCPU a 24 vCPU.

L'input utilizzato per effettuare l'analisi è stato:

- N = 5000
- I = 50

Numero di Processi	Tempo(s)	Speedup
1	27.242525	1.00
2	20.424715	1.33
3	13.919035	1.95
4	11.126861	2.44
5	9.303328	2.92
6	7.762574	3.51
7	6.550036	4.15
8	5.783624	4.71
9	5.238248	5.20
10	4.806311	5.67
11	4.596453	5.93
12	4.669662	5.84
13	4.017718	6.79
14	3.678917	7.42
15	3.514501	7.76
16	3.336226	8.18
17	3.172971	8.59
18	3.041431	8.96
19	2.980639	9.14
20	3.128735	8.73

Numero di Processi	Tempo(s)	Speedup
21	2.704358	10.08
22	2.671802	10.20
23	2.584825	10.55



Scalabilità Debole

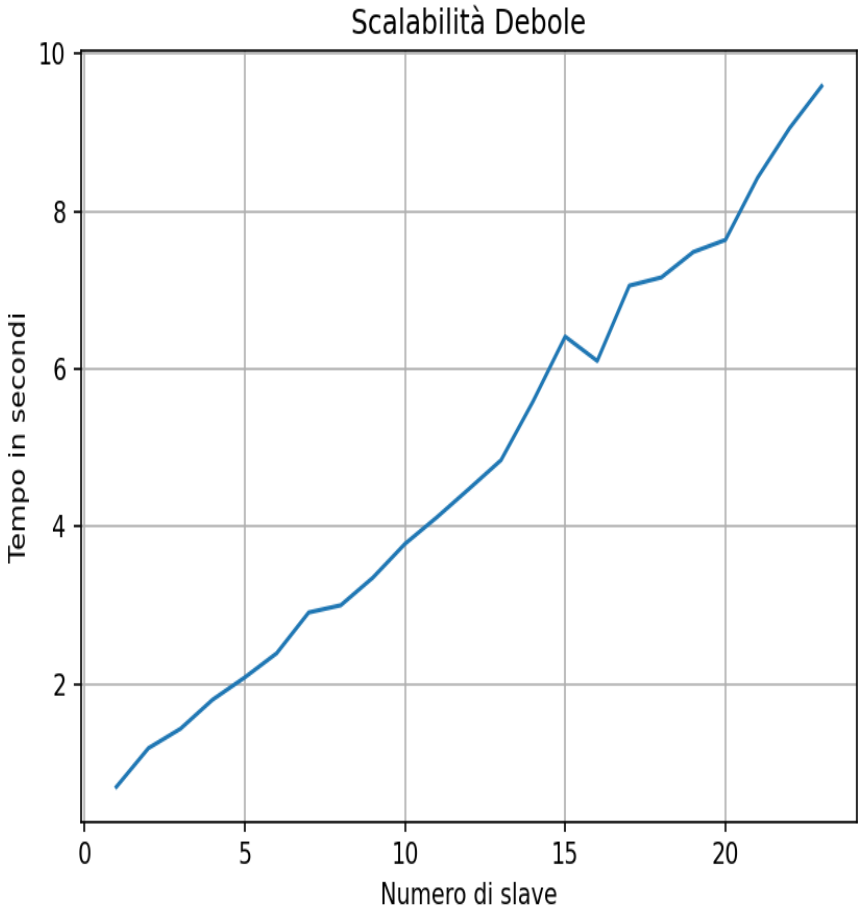
Anche la scalabilità debole è stata ottenuta analizzando più esecuzioni dell'algoritmo sul Cluster.

L'input utilizzato per effettuare l'analisi è stato:

- N = 400*np
- I = 50

Numero di Processi	Tempo(s)	Dimensione Input(N)	Dimensione Input(I)
1	0.698966	400x400	50
2	1.189706	800x800	50
3	1.436100	1200x1200	50
4	1.804403	1600x1600	50
5	2.086154	2000x2000	50

Numero di Processi	Tempo(s)	Dimensione Input(N)	Dimensione Input(I)
6	2.394085	2400x2400	50
7	2.911704	2800x2800	50
8	3.003367	3200x3200	50
9	3.353309	3600x3600	50
10	3.781323	4000x4000	50
11	4.120756	4400x4400	50
12	4.480034	4800x4800	50
13	4.845016	5200x5200	50
14	5.592357	5600x5600	50
15	6.413034	6000x6000	50
16	6.103976	6400x6400	50
17	7.058765	6800x6800	50
18	7.163511	7200x7200	50
19	7.486568	7600x7600	50
20	7.639772	8000x8000	50
21	8.427413	8400x8400	50
22	9.056979	8800x8800	50
23	9.590614	9200x9200	50



Conclusioni

Analizzando i benchmark e nello specifico i risultati ottenuti in termini di scalabilità forte e scalabilità debole possiamo notare che l'introduzione del parallelismo ha portato un notevole vantaggio, osservabile analizzando lo speedup,tenendo conto ovviamente dell'overhead introdotto dalla comunicazione tra i processi, e dalla latenza introdotta dall'utilizzo di un cluster.