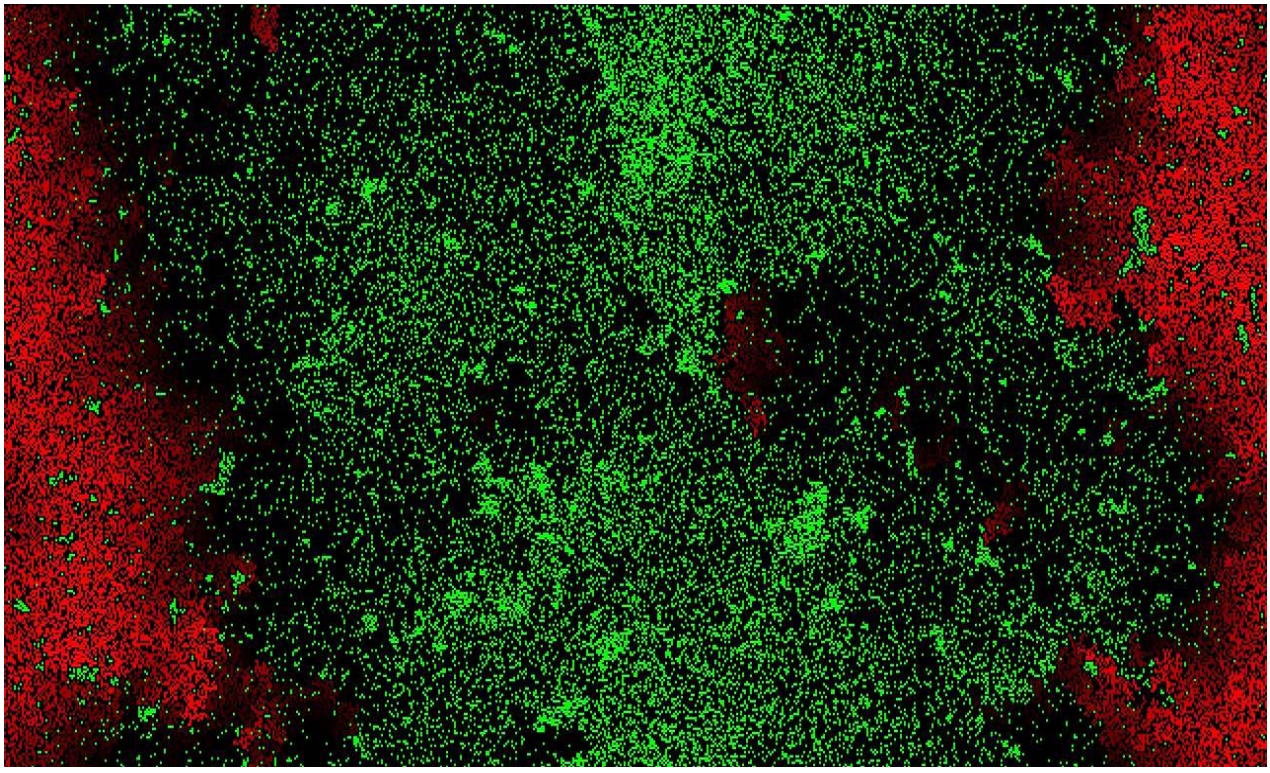
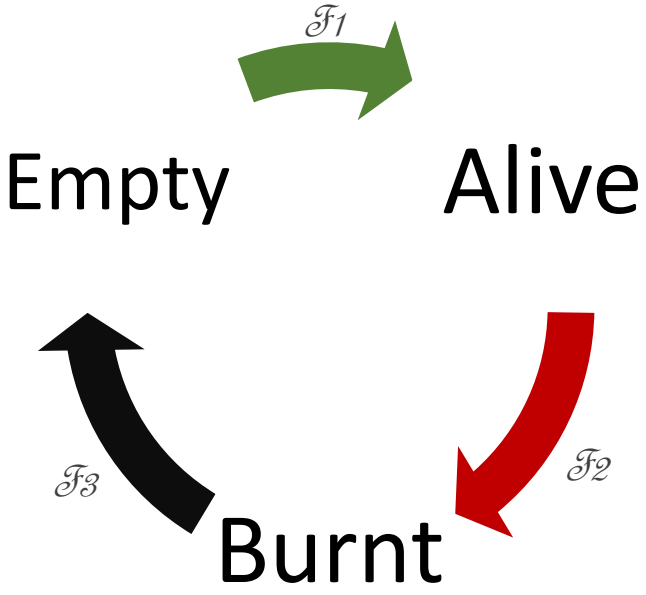
Progetto MPI Giovanni Celia: Forest Fire.

Un automa è in generale un sistema o modello dinamico discreto che assume determinati comportamenti indipendentemente dall'istante di tempo in cui agisce, i quali sono descritti da regole bene definite. In maniera un po' più formale possiamo descrivere un automa generico (ne esistono più tipi) come un insieme di segnali in entrata (ha dunque in suo alfabeto) e segnali in uscita, i quali variano in base allo stato in cui si trovano. In particolare, un automa cellulare può sempre essere descritto come sopra riportato, aggiungendo che la sua rappresentazione visiva in output è una griglia formata da celle di una qualsiasi forma. ***Forest Fire*** *è un tipo di automa cellulare, la sua forma in output è in genere la seguente:*



Il numero totale di stati dell'automa cellulare rappresentato qui sopra è 3: Empty, Alive, Burnt. Possiamo facilmente descrivere il suo comportamento nella figura qui sotto:



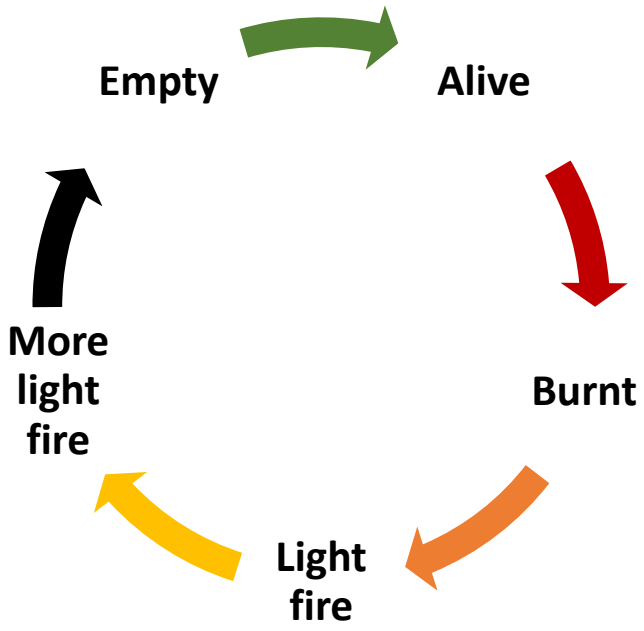
Lo stato iniziale in cui ogni cella si trova è lo stato Empty. Si può passare da uno stato all'altro tramite delle regole fissate che sono:

F1: La cella vuota ha probabilità 2/n di fare nascere un albero;

*F2: Un albero può bruciare se una delle celle adiacenti è in fiamme (8 celle adiacenti) oppure se un fulmine cade sulla stessa con probabilità 2/n*1000;*

F3: Un albero in fiamme deve necessariamente divenire nuovamente una cella vuota nella generazione successiva.

Ovviamente le probabilità, pur mantenendo più o meno questa proporzione, possono variare. Tramite queste semplici regole, questo automa, nel tempo, è capace di simulare in linea generica la propagazione degli incendi nelle foreste. Esistono vari algoritmi che vanno a tenere conto anche della direzione del vento, intensità di fuoco ed altri particolari, ma la sostanza non cambia, al più si aggiungono degli stati supplementari per gestire altre casistiche, ma questo rimane lo schema base dal quale non si può prescindere. Nell'algoritmi che presenterò si aggiungono 2 stati supplementari che sono 2 intensità di fuoco più lievi rispetto allo stato Burnt, non influiscono in maniera eccessiva nell'algoritmo base, ma danno la possibilità agli alberi di crescere più lontani dalle celle che hanno appena preso fuoco, in quanto dopo lo stato Burnt avremo uno stato in cui la cella si presenta di colore arancione ed infine uno stato in cui la cella si presenta di colore giallo; in entrambi i casi la generazione successiva non potrà generare un albero, ma dovrà essere una cella vuota necessariamente. Il diagramma dunque diventa il seguente:



Come si può notare gli stati aggiunti sono Light fire e More light fire, i quali passano allo stato successivo a senza avere né vincoli né altre regole particolari. L'algoritmo che andrò ora a presentare può essere diviso in due "parti": grafica e logica. Per la grafica ho utilizzato la libreria Allegro5 mentre per la logica ho fatto dei test di efficienza usando la libreria mpi.h e settando il numero di processori usati a 1(seriale), 2, 4 e 6(massimo processori fisici).

Il codice

*Cominciamo con la funzione **generaAlbero(…)** e **generaFulmine(…)** che hanno comportamenti molto simili:*

```
bool generaAlbero(int rank){

    int k=1000; //1000
    int n = rand()%k+1 + rank;
    if(n == rand()%k+1  || n == rand()%k+1 ) {
        return true;
    }
    return false;

}
```

```
bool generaFulmine(int rank){

    int k=1000000; //1000000
    int n = rand()%k+1+ rank;

    if(n == rand()%k+1 || n == rand()%k+1){
        return true;
    }
    return false;

}
```

Essenzialmente quello che fanno è andare a calcolare la probabilità di generare un fulmine o un albero, rispettivamente 2/1000000 e 2/1000, generando due numeri random n, sommandogli il numero del processo che genera le funzioni, e controllando se questo numero è uguale ad un degli altri due numeri random generati.
*La prossima funzione **alberoVicinoInFiamme(…)** è di fondamentale importanza, in quando sarà quella che determinerà l'effettiva espansione dell'incendio nella foresta:*

```
bool alberoVicinoInFiamme( int i, int j, int myDim, int M[][dim], int upVector
[dim], int downVector [dim]){
    bool b=false;
    if(i==0){
        if(j==0){
            for(int k=i; k<=i+1; k++){
                for(int h=j; h<=j+1; h++){
                    if(k!=i || h!=j ){
                        if(M[k][h]==2){
                            b=true;
                            return b;
                        }
                    }
                }
            }
            for(int v=j; v<=j+1; v++){
                if(upVector[v]==2){
                    b=true;
                    return b;
                }
            }
        }
        else if(j==dim){
            for(int k=i; k<=i+1; k++){
                for(int h=j-1; h<=j; h++){
                    if(k!=i || h!=j ){
                        if(M[k][h]==2){
```

```
                        b=true;
                        return b;
                    }
                }
            }
        }
        for(int v=j; v<=j+1; v++){
            if(upVector[v]==2){
                b=true;
                return b;
            }
        }
    }
    else{
        for(int k=i; k<=i+1; k++){
            for(int h=j-1; h<=j+1; h++){
                if(k!=i || h!=j ){
                    if(M[k][h]==2){
                        b=true;
                        return b;
                    }
                }
            }
        }
        for(int v=j; v<=j+1; v++){
            if(upVector[v]==2){
                b=true;
                return b;
            }
        }
    }
}
else if(i==myDim){
    if(j==0){
        for(int k=i-1; k<=i; k++){
            for(int h=j; h<=j+1; h++){
                if(k!=i || h!=j ){
                    if(M[k][h]==2){
                        b=true;
                        return b;
                    }
                }
            }
        }
        for(int v=j; v<=j+1; v++){
            if(downVector[v]==2){
                b=true;
```

```
                    return b;
                }
            }
        }
        else if(j==dim){
            for(int k=i-1; k<=i; k++){
                for(int h=j-1; h<=j; h++){
                    if(k!=i || h!=j ){
                        if(M[k][h]==2){
                            b=true;
                            return b;
                        }
                    }
                }
            }
            for(int v=j; v<=j+1; v++){
                if(upVector[v]==2){
                    b=true;
                    return b;
                }
            }
        }
        else{
            for(int k=i-1; k<=i; k++){
                for(int h=j-1; h<=j+1; h++){
                    if(k!=i || h!=j ){
                        if(M[k][h]==2){
                            b=true;
                            return b;
                        }
                    }
                }
            }
            for(int v=j; v<=j+1; v++){
                if(upVector[v]==2){
                    b=true;
                    return b;
                }
            }
        }
    }
    else if(j==0){

        for(int k=i-1; k<=i+1; k++){
            for(int h=j; h<=j+1; h++){
                if(k!=i || h!=j ){
                    if(M[k][h]==2){
```

```
                        b=true;
                        return b;
                    }
                }
            }
        }
    }
    else if(j==dim){

        for(int k=i-1; k<=i+1; k++){
            for(int h=j-1; h<=j; h++){
                if(k!=i || h!=j ){
                    if(M[k][h]==2){
                        b=true;
                        return b;
                    }
                }
            }
        }
    }
    else{
        for(int k=i-1; k<=i+1; k++){
            for(int h=j-1; h<=j+1; h++){
                if(k!=i || h!=j ){
                    if(M[k][h]==2){
                        b=true;
                        return b;
                    }
                }
            }
        }
    }
    return b;
}
```

Questa funzione si va a richiamare quando su una cella è presente un albero ovviamente, quello che si va a controllare è se una cella adiacente è un albero in fiamme, se lo è allora la variabile booleana b passa a true e ritorniamo il valore. La funzione è molto lunga a causa dei vari controlli da effettuare in base alla posizione (i, j) nella matrice, infatti avremo i seguenti casi:

- *i=0, j=0 (ci troviamo nella prima cella a sinistra)*
- *i=0, j=dim (prima riga, ultima colonna)*
- *i=0, j>0 & j<dim (prima riga, qualsiasi colonna diversa da 0 e dim)*
- *i=myDim, j=0 (ultima riga, prima colonna)*
- *i=myDim, j=dim (ultima riga, ultima colonna)*
- *i=myDim, j>0 & j<dim (ultima riga, qualsiasi colonna diversa da 0 e dim)*
- *i>0 & i<myDim, j=0 (prima colonna, qualsiasi riga diversa da 0 e myDim)*
- *i>0 & i<myDim, j=dim (ultima colonna, qualsiasi riga diversa da 0 e myDim)*

*In base al caso in cui ci troviamo dobbiamo controllare se la riga che ci serve per valutare il prossimo stato rientra nella nostra sottomatrice oppure no, nel caso in cui non fosse così la funzione prende come argomenti **upVector** e **downVector**, nient'altro che degli array che contengono delle copie dell'ultima riga del processo precedente e della prima riga del processo successivo. Queste copie sono scmabiate tra i processi in maniera esplicita tramite delle **Send(…)** e delle **Recv(…)** come illustrato qui sotto:*

```
if(rank == 0){
        MPI_Send(mySubset[0], dim, MPI_INT, sizeWorld-1, 8, MPI_COMM_WORLD);
        MPI_Send(mySubset[myDim-1], dim, MPI_INT, rank+1, 8, MPI_COMM_WORLD);
        MPI_Recv(upVector, dim, MPI_INT, sizeWorld-
1, 8, MPI_COMM_WORLD, &status);
        MPI_Recv(downVector, dim, MPI_INT, rank+1, 8, MPI_COMM_WORLD, &status)
;
    }
    else if(rank == sizeWorld-1){
        MPI_Send(mySubset[0], dim, MPI_INT, rank-1, 8, MPI_COMM_WORLD);
        MPI_Send(mySubset[myDim-1], dim, MPI_INT, 0, 8, MPI_COMM_WORLD);
        MPI_Recv(upVector, dim, MPI_INT, rank-1, 8, MPI_COMM_WORLD, &status);
        MPI_Recv(downVector, dim, MPI_INT, 0, 8, MPI_COMM_WORLD, &status);
    }
    else {
        MPI_Send(mySubset[0], dim, MPI_INT, rank-1, 8, MPI_COMM_WORLD);
        MPI_Send(mySubset[myDim-1], dim, MPI_INT, rank+1, 8, MPI_COMM_WORLD);
        MPI_Recv(upVector, dim, MPI_INT, rank-1, 8, MPI_COMM_WORLD, &status);
        MPI_Recv(downVector, dim, MPI_INT, rank+1, 8, MPI_COMM_WORLD, &status)
;
    }
```

Come si può notare, è come se la matrice sia "circolare" in senso verticale, in quanto processo Master e l'ultimo Slave, si passano le rispettive righe necessarie. Infine, abbiamo la funzione di transizione che calcola per ogni cella della nostra porzione di matrice cosa inserire nella nuova porzione:

```
for (int i = 0; i < myDim; i++) {
        for (int j = 0; j < dim; j++) {
            if(tempSubset[i][j]== 0){
                if(generaAlbero(rank))
                    currentSubset[i][j] = 1;
                else
                    currentSubset[i][j] = tempSubset[i][j];
            }
            else if(tempSubset[i][j]==1){
                if(alberoVicinoInFiamme( i, j, myDim, tempSubset, upVector, do
wnVector) || generaFulmine(rank))
                        currentSubset[i][j]= 2;
                else
                    currentSubset[i][j] = tempSubset[i][j];
            }
            else if(tempSubset[i][j]==2)
                        currentSubset[i][j]=8;

            else if(tempSubset[i][j]==8)
                        currentSubset[i][j]=9;

            else if(tempSubset[i][j]==9)
                        currentSubset[i][j]=0;
        }
    }
```

Per 0 si intende lo stato di Empty, 1 Alive, 2 Burnt, 8 Light fire, 9 More light fire.
*La complessità computazionale in fatto di operazioni eseguite è asintoticamente equivalente a O(n^2*k), dove n indica una dimensione della matrice e k indica il numero di generazioni (da notare che se k<n*n, non è influente a livello asintotico).*
Effettuando dei test per quanto riguarda l'esecuzione del programma utilizzando un solo processo e più processi su 1000, 10.000, 100.000 generazioni con una matrice 500x500 ho ottenuto i seguenti risultati:

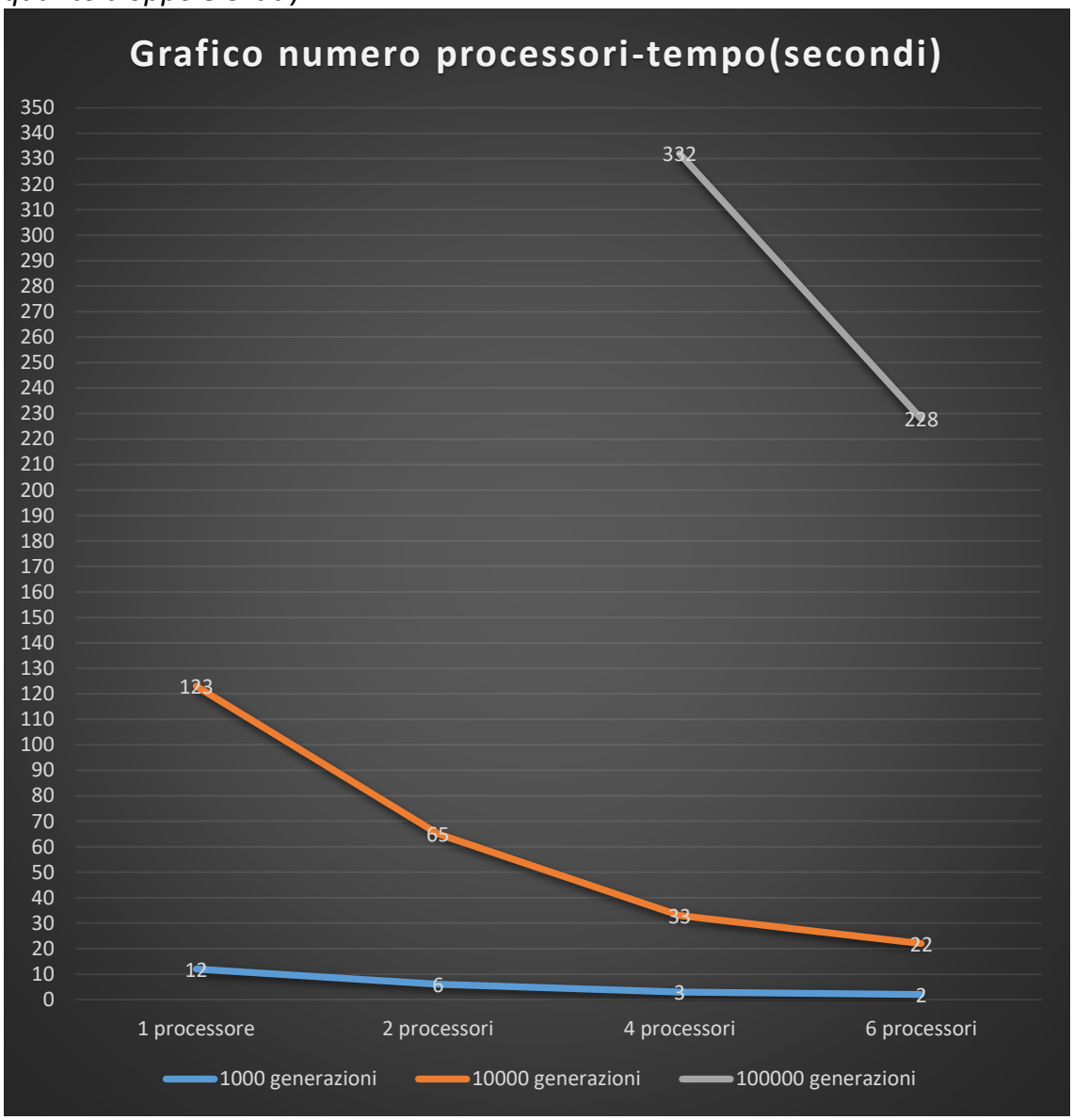
Tempo in secondi	1000	10.000	100.000
1 processore	12	123	23 minuti*
2 processori	6	65	11 minuti*
4 processori	3	33	332
6 processori	2	22	228

(*) indicano valori calcolati approssimativamente seguendo lo stesso indice Ts/Tp, in quanto i tempi di attesa erano >10 minuti.

I valori di Ts/Tp ottenuti hanno una media abbastanza costante di

- *2 per 2 processori utilizzati;*
- *4 per 4 processori utilizzati;*
- *6 per 6 processori utilizzati.*

Andiamo ora a vedere il grafico (dove ho omesso i valori 23 minuti e 11 minuti in quanto troppo elevati).



*Come possiamo notare, la parallelizzazione ha prodotti ottimi risultati anche solo per il fatto riuscire a terminare il programma con 100.000 cicli in tempistiche accettabili date le dimensioni della matrice (100.000*500*500 = 25 miliardi di operazioni circa); in generale abbiamo di un incremento delle prestazioni 4 volte maggiori utilizzando 6 processori rispetto ad 1 singolo processore, c'è da tenere conto anche dello sforzo sulla CPU che aumenta del 9-10% per ogni processore usato. La totalità dei test è stata effettuata con un* ***Ryzen 5 2600X 4.2 GHz RAM 2x8 DDR4 CL15 3000 MHz****.*