Relazione Progetto APSD: War Simulator

Jacopo Garofalo

15 marzo 2023

Indice

1 Introduzione

Il progetto realizzato descrive uno scenario particolare di uno scontro tra due popolazioni, rappresentato tramite un automa cellulare, che va a richiamare una sorta di modello *preda-predatore*, poichè ogni individuo funge sia da potenziale che potenziale predatore.
Ogni individuo ha come scopo quello di riuscire a crescere e sopravvivere alla guerra fino alla sua morte di vecchiaia, sconfiggendo durante la sua vita la popolazione nemica.
Tutta l'idea dell'automa cellulare è stata ideata da me, assieme alla sua struttura e logica, quindi potrebbero esserci piccole imprecisioni, mentre la parte riguardante la visualizzazione a schermo dell'automa cellulare è stata implementata facendo uso della libreria grafica **Allegro5**, pensata appositamente per il linguaggio **C/C++**.

2 Descrizione dell'idea

L'idea implementata descrive un possibile scenario di combattimento che potrebbe avvenire tra due popolazioni. Come in una vera guerra, lo scopo ultimo è quello di riuscire a sopravvivere nel caos generato dalla guerra e sopraffare i nemici, andando quindi a simulare una lotta che durerà per tutta la durata della vita delle celle. Le celle possono quindi assumere 2 diversi macro-stati differenti:

- **Attaccante**.
- **Difensore**.

i quali a loro volta avranno 2 importanti sotto-stati differenti:

- **Bambino**: rappresenta una cella appena nata o creata da poco.
- **Adulto**: rappresenta una cella creata da un pò di tempo.
- **Morto**: rappresenta una cella che è stata uccisa per uno dei motivi presenti nella logica dell'automa.

Tutti questi stati sono stati definiti nella struttura interna di ogni cella assieme a qualche altro dato utilizzato per definire meglio la "condizione di vita" di ogni singola cella.
Le varie celle combatteranno tra di loro per poter sopravvivere e far sopravvivere la popolazione di cui fanno parte, crescendo nel frattempo se si trattano di celle "bambine", morendo se vengono in qualche modo sopraffatte da celle

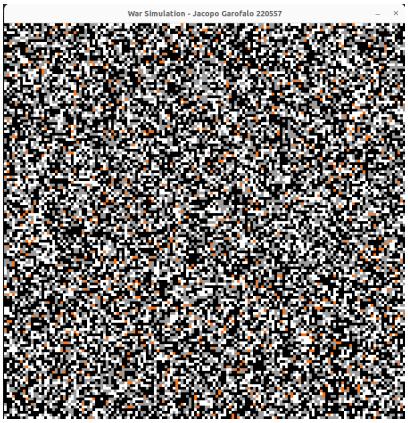

Figura 1: Esempio di possibile iterazione del programma.

appartenenti alla popolazione avversarie oppure quando raggiungono il limite di durata della loro vita.
Una volta morte, le celle torneranno in vita una volta passato un certo periodo di tempo e ricominceranno la lotta per la sopravvivenza.

3 Implementazione

Per poter descrivere in maniera efficace questo modello di automa cellulare, si tratterà prima la struttura implementativa come se fosse sequenziale, in seguito verrà introdotta la parallelizzazione, prima come se venisse fatta con una topologia uni-dimensionale e poi bi-dimensionale, andando ad illustare dunque quali tipi di dati si è utilizzati.

3.1 Strutture dati usate

Per implementare l'automa cellulare alla base del progetto del progetto, si è optato per un **vettore dinamico** per favorire l'efficienza avendo così i dati contigui in memoria, che per comodità di utilizzo e soprattutto per semplicità di lettura e escrittura del codice, verrà trattato in tutto il codice del programma come se fosse una matrice grazie ad una funzione che si occuperà di convertire le coordinate matriciali in un indice per accedere alla corrispondente posizione del vettore:

```
int at(int r, int c) { return r*Cols+c; }
```

Si è optato per questa scelta seguendo l'esempio fatto dai professori durante i vari esercizi svolti in aula, avendo una struttura dei movimenti in più direzioni, come si potrà osservare nella parte riguardante la parallelizzazione.
Da qui in poi parleremo di matrice per facilità esplicativa, ma si alluderà sempre al nostro vettore dinamico.

3.2 Struttura delle celle

Per strutturare le singole celle si è pensata ad una soluzione particolare, poichè per poter rappresentare la complessa struttura delle celle sarebbe servita una **struct**, la quale avrebbe reso l'automa particolarmente pesante a livello di memoria, oltre a intaccare le performance. Perciò, si è usata la **union**, messa a disposizione dal linguaggio C, per non appesantire troppo l'automa.
Ogni cella deve avere a disposizione un totale minimo di 10 bit per poter essere rappresentata, quindi si è dovuto fare in modo che fosse rappresentata tramite uno **short**, il quale mette a disposizione ben 16 bit.
Di questi 10 bit, i primi 2 servono per capire a quale popolazione appartiene **al momento** la cella, i successivi 2 serviranno per capire lo schieramento originale (*verrà implementato in una futura versione dell'automa*), i seguenti 2 invece vengono usati per identificare meglio la cella in sè, quindi se essa è una cella bambina oppure se è morta, mentre i restanti 3 sono usati per monitorare il tempo di vita della cella nello stato raggiunto, tempo che verrà azzerato ogni qualvolta la cella passa in un nuovo stato:

- se **attacker** è 1 allora appartiene agli attaccanti.
- se **defender** è 1 allora appartiene ai difensori.
- se **isChild** è 1 allora è una cella bambina.
- se **isDead** è 1 allora è una cella morta.

3.3 Il modello

Come già detto prima, il modello su cui si basa l'implementazione è quello di un automa cellulare, mentre il vicinato verrà controllato tramire il **vicinato di Moore**, quindi si andranno a prendere in considerazione tutte i possibili vicini della singola cella in tutte le direzioni per poter calcolare il nuovo stato della cella presa in questione.
Tutto il processo si svolge facente uso di 2 matrici, una di lettura e una di scrittura, e procede come segue:

1. Viene letta una cella dalla matrice di lettura.

2. Viene calcolata la probabilità di sopravvivenza in base ai vicini, ed in base a questa la cella morirà o sopravviverà, per poi essere riscritta nella matrice di scrittura.

3. Si ripete il punto 1 fino a quando non si è iterata tutta la matrice di lettura.

4. Avviene uno swap tra le 2 matrici, cosicchè la matrice di scrittura venga considerata la nuova matrice di lettura e possa ricominciare il ciclo.

Questo processo verrà fatto un numero N di volte, ovvero il cosiddetto numero di step che l'automa può fare.

4 Parallelizzazione

Per rendere il programma parallelo, ovvero fare in modo che il programma venga eseguito *contemporaneamente* su più processori, in particolare sui vari cores della cpu, si è utilizzato il paradigma del message passing, quindi più processori eseguono in maniera concorrente una loro copia del programma, dove ogni processo opera sulla propria memoria locale e quindi non potrà accedere alle risorse dell'altro. Il message passing consiste quindi nella comunicazione tra i vari processi tramite lo scambio di messaggi.
Per fare ciò si è optato per l'utilizzo della libreria MPI, che fornisce all'utente un'interfaccia per gestire il message passing. Ogni processo avrà dunque una sua sotto-matrice per poter lavorare autonomamente, evitando quindi il classico *master-slave* che impone che ci sia un processo principale che detterà agli altri processi cosa bisogna fare, anche se va detto che a causa del fatto che la libreria grafica **Allegro5** permette la stampa solo ad un processo, si è dovuto implementare un sistema dove un processo scompone la matrice che dovrà essere stampata ai vari processi, i quali dopo che avranno finito di lavorarci rimanderanno il proprio pezzo al processo addetto alla stampa. Tale sistema è stato utilizzato solamente per dare una rappresentazione grafica dell'automa, infatti in condizioni normali l'automa non ne necessita.

4.1 Implementazione parallela

Partendo dagli **MPI Datatype** usati all'interno del nostro programma, si possono notare la presenza di ben 3 tipi diversi, necessari per poter effettivamente realizzare la comunicazione nelle direzioni a cui siamo interessa-

ti, ovvero in orizzonatale (nord-sud), in verticale (est-ovest) e in diagonale (nord-est, sud-ovest, etc.):

- **columnType**: usato dai processi per mandare le colonne della propria sotto-matrici ai vicini interessati, comunicando quindi in orizzontale.
- **rowType**: usato dai processi per mandare le righe della propria sotto-matrice ai vicini interessati, comunicando quindi in verticale.
- **cornerType**: usato dai processi per mandare i *corner*, ovvero i vertici della propria sotto-matrice, ai vicini interessati, comunicando quindi in diagonale.

Inoltre, in aggiunta a questi primi datatype, ne sono stati creati altri 2, usati **esclusivamente** per la stampa per il motivo spiegato poco prima:

- **innerGridType**: usato dai processi per identificare la propria intera sotto-matrice.
- **contiguousGridType**: usato dal processo addetto alla stampa. Avrà gli stessi elementi di innerGridType, ma disposti in modo diverso cosicchè possano essere contigui in memoria.

Per quanto riguarda la creazione in sè, si è proceduto in questo modo:

```
Dimensione totale della sotto-griglia:
const int outer_sizes[] = { my_rows, my_cols };

Dimensione della sotto-griglia senza celle-halo:
const int inner_sizes[] = { my_inner_rows, my_inner_cols };

Partenza da (0, 0) per semplificare i calcoli.
const int starts[] = { 0, 0 };

MPI_Type_create_subarray(2, outer_sizes, inner_sizes,
    starts, MPI_ORDER_C, MPI_UINT16_T, &inner_grid_t);

Poiche inner_grid_size = my_inner_rows * my_inner_cols,
avranno quindi la stessa quantità di innerGridType.
MPI_Type_contiguous(inner_grid_size, MPI_UINT16_T,
    &contiguousGridType);

MPI_Type_vector(my_inner_rows, 1, my_cols,
```

```
    MPI_UINT16_T, &columnType);

MPI_Type_vector(1, my_inner_cols, my_cols,
    MPI_UINT16_T, &rowType);

MPI_Type_vector(1, 1, my_cols, MPI_UINT16_T, &cornerType);
```

Per simulare la posizione relativa tra i processi ho usato un comunicatore cartesiano su due dimensioni apposito denominato **myComm**:

```
int dims[2] = { cfg->y_threads, cfg->x_threads };
int periods[2] = { 0, 0 };
MPI_Cart_create(MPI_COMM_WORLD, 2, dims,
    periods, 0, &cave_comm);
```

In questo modo è stato relativamente facile conoscere per esempio chi fosse il thread in alto a destra rispetto a questo thread, facilitando le comunicazioni con i thread vicini.

4.2 Suddivisione Griglia

Come accennato prima il processo principale genera l'intera griglia in modo pseudo-casuale partendo da un seed. Una volta generata, la suddivide in blocchi di uguale grandezza e li invia a tutti i thread, se stesso compreso.

```
Qua il processo principale suddivide la griglia iniziale
per tutti i processi (se stesso incluso).
void scatter_initial_grid() {
    uint16_t* dest_buff = &read_grid[my_cols + 1];
    MPI_Scatter(root_grid, 1, contiguousGridType,
    dest_buff, 1, innerGridType, ROOT_RANK, myComm);
}
```

Arrivati a questo punto, entrano in gioco i 3 **MPI Datatype** pensati per lo scambio bordi/corner, il quale verrà svolto in modo **non-bloccante**, cosi da evitare attese sui messaggi.
Ad esempio, l'invio delle colonne funzionerà così:

```
void send_columns() {
    if(neighbours_ranks[MIDDLE][LEFT] != MPI_PROC_NULL) {
        MPI_Request req;
        int start_idx = my_cols + 1;
        MPI_Isend(&read_grid[start_idx], 1, columnType,
        neighbours_ranks[MIDDLE][LEFT],
        1001, myComm, &req);
        MPI_Request_free(&req);
    }

    if(neighbours_ranks[MIDDLE][RIGHT] != MPI_PROC_NULL) {
        MPI_Request req;
        int start_idx = my_cols + my_inner_cols;
        MPI_Isend(&read_grid[start_idx], 1, columnType,
        neighbours_ranks[MIDDLE][RIGHT],
        1002, myComm, &req);
        MPI_Request_free(&req);
    }
}
```

4.3 Rendering

Quando tutti i processi hanno finito di aggiornare la propria parte di griglia sorge il problema del rendering, Allegro infatti non da la possibilità di modificare la grafica da più threads, quindi c'è il bisogno di raggruppare nuovamente tutte le sotto-griglie nel processo principale, così da far gestire la grafica ad un solo processo.
Sempre per la questione detta prima, si passa dal tipo **innerGridType** a **contiguousGridType**.

```
void gather_grid() {
    uint16_t* send_buff = &read_grid[my_cols + 1];
    MPI_Gather(send_buff, 1, innerGridType, root_grid, 1,
    contiguousGridType, ROOT_RANK, myComm);
}
```

Al termine di questa funzione, il processo principale si ritroverà all'interno di `root_grid` tutte le sotto-griglie in modo contiguo, senza celle-halo di troppo e pronta per essere mostrato all'utente.

5 Misure e tempi

Le misurazioni sono state eseguite sulla seguente architettura:
CPU AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz, *16,0 GB RAM*, *GPU Nvidia GeForce GTX 1660 Ti*. I principali valori calcolati sono stati:

- **tempo di esecuzione**;
- **speedup**: indica il rapporto tra il tempo di esecuzione seriale e il tempo di esecuzione parallelo;
- **efficienza**: indica il rapporto tra speedup e numero di processori usati.

I tempi sono stati presi in base ai seguenti criteri:

- *numero di threads*: il programma è stato eseguito con 1, 2, 3, 4, 6, 5 e 8 threads;
- *dimensione* della matrice: sono state scelte le dimensioni 600x600, 900x900 e 1200x1200;
- *steps*, ovvero numero di iterazioni compiute: le misure sono state prese su 100 steps.

Tutte le misure sono state prese disabiltando la componente grafica, dato che avrebbe alterato enormemente queste misure e non serviva quindi ai fini dell'esperimento (*si parla di un circa 90%*). Nelle immagini si può notare una sorta di griglia rossa che mostra come effettivamente la matrice è stata suddivisa tra i processi.

Processi	1	2	3	4	6	8
Tempo	1.312	0.685	0.463	0.354	0.285	0.227
Speed-Up	1	1.915	2.833	3.706	4.603	5.779
Efficienza	1	0.957	0.944	0.926	0.767	0.722

Tabella 1: Matrice 600x600

Processi	1	2	3	4	6	8
Tempo	3.027	1.501	1.021	0.749	0.634	0.489
Speed-Up	1	2.016	2.964	4.041	4.774	6.190
Efficienza	1	1.008	0.988	1.010	0.795	0.773

Tabella 2: Matrice 900x900

Processi	1	2	3	4	6	8
Tempo	5.372	2.660	1.790	1.362	1.126	0.861
Speed-Up	1	2.019	3.001	3.944	4.770	6.239
Efficienza	1	1.009	1.000	0.986	0.795	0.779

Tabella 3: Matrice 1200x1200

5.1 Considerazioni

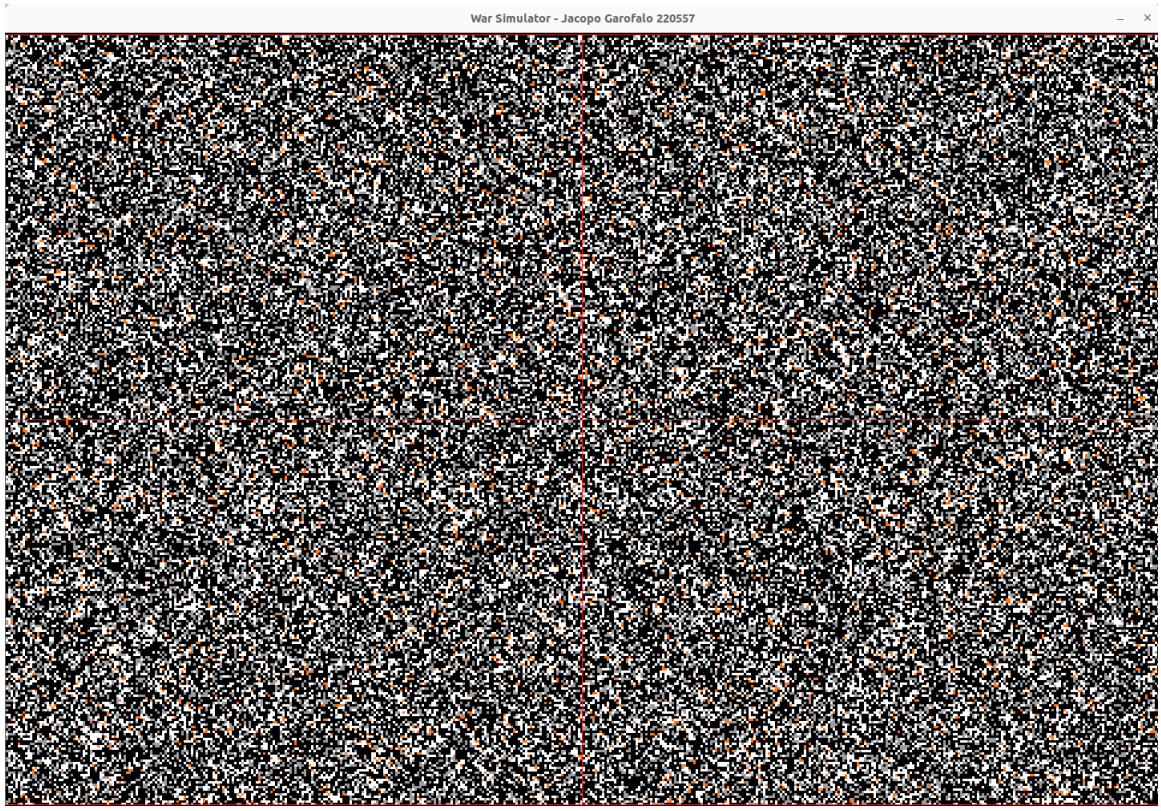

Figura 2: Esempio di esecuzione con 4 processi.

Si può notare quasi sempre un ridimensionamento del tempo di esecuzione quando vengono incrementati i numeri di processi in esecuzione, permettendo quindi di osservare come lo speed-up al crescere delle dimensioni del problema tende ad aumentare mentre l'efficienza tende a rimanere costante tranne nel caso di 6 e 8 processi, dove le comunicazioni tra i processi aumentano di molto, portando dunque molto **overhead**.
Inoltre, in alcuni casi possiamo notare che l'efficienza tende ad essere super-

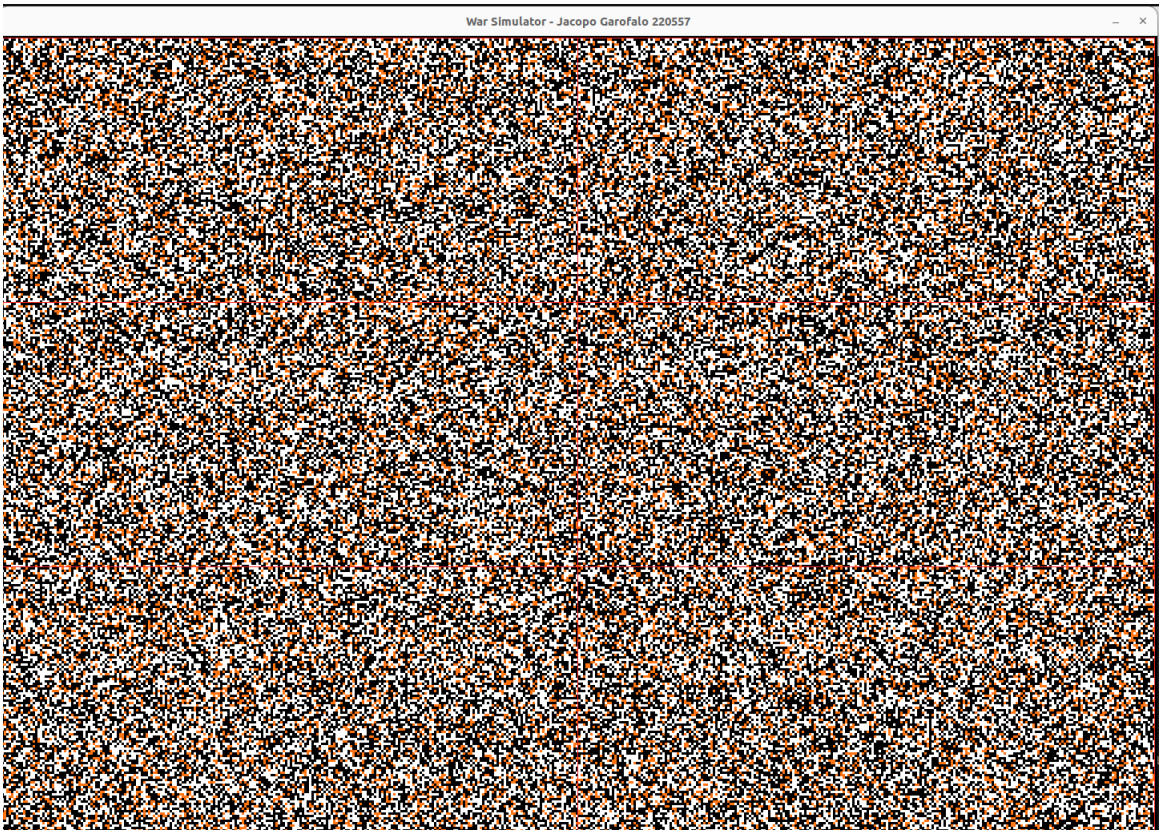

Figura 3: Esempio di esecuzione con 6 processi.

lineare, dovuto sia alla buona scalabilità, sia al buon ridimensionamento del problema, poichè in alcuni casi si è osservato come giovi molto ai tempi di esecuzione.