UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di laurea magistrale in Sicurezza Informatica

Progetto programmazione concorrente, parallela e su Cloud

Anno Accademico 2020/2021

Benedetto Sommese

Sommario

Capitolo 1: Game of life

All'interno di questo documento andremo a parlare dell'implementazione di Game of life e successivamente andremo ad analizzare le prestazioni sulle macchine "t2.2xlarge".

1.1: Introduzione

The game of life è un gioco senza giocatori che non richiede ulteriori input. All'interno di game of life si crea una configurazione iniziale osservando come si evolve. L'universo di Game of life è una griglia ortogonale bidimensionale infinita di celle quadrate, ognuna delle quali si trova in uno dei due possibili stati: vivo o morto. È possibile utilizzare una matrice di boolean per determinare se la cellula è viva o morta. Ogni cella interagisce con i suoi otto vicini che sono le celle adiacenti orizzontalmente, verticalmente o diagonalmente. Ad ogni passaggio nel tempo, si verificano le seguenti transizioni:

- qualsiasi cellula viva con meno di 2 vicini vivi muore come se fosse causata dalla sottopopolazione;
- qualsiasi cellula viva con 2 o 3 vicini vivi sopravvive alla generazione successiva;
- qualsiasi cellula viva con più di 3 vicini vivi muore per sovrappopolazione;
- qualsiasi cellula morta con esattamente 3 vicini vivi diventa una cellula viva per riproduzione.

Il modello iniziale costituisce il seme del sistema e la prima generazione viene creata applicando le regole di sopra simultaneamente a ogni cellula del seme: nascite e morti avvengono simultaneamente e il momento discreto in cui ciò accade è talvolta chiamato segno di spunta (ogni generazione è una pura funzione della precedente). Inoltre, le regole continuano ad essere applicate ripetutamente per creare nuove generazioni.

Il programma deve essere in grado di simulare il processo per un numero di passi N. Il processo master inizializza una matrice di boolean a caso e la suddivide tra i P processori. Il processo master nel nostro caso contribuisce al calcolo. Ogni slave simula il gioco e invia le celle fantasma corrispondenti ai processori del suo vicino per il passaggio alla simulazione successiva. La parte difficile del problema consiste nel suddividere equamente la matrice tra i processori. Per fare ciò, noi effettueremo la suddivisione sfruttando le righe della matrice. Inoltre, il programma deve funzionare su qualsiasi dimensione di matrice e numero di processi.

1.2: Descrizione implementazione

In questo paragrafo vedremo come è stata effettuata l'implementazione della soluzione.

1.2.1: Parametri e comando per eseguire

La soluzione proposta prende in input come parametro il numero di processi che andranno ad operare in simultanea per la soluzione del problema, il numero di iterazioni da eseguire le quali determinano il numero di generazioni che si saranno, il numero di righe di cui sarà composta la matrice generata e il numero di colonne di cui sarà composta la stessa.

```
int iterazioni = atoi(argv[1]); //recuperiamo il numero di iterazioni da effettuare
int righe = atoi(argv[2]); //recuperiamo il numero di righe
int colonne = atoi(argv[3]); //recuperiamo il numero di colonne
bool **matrice = alloc_array_bool(righe, colonne); //allocazione dinamica della matrice di boolean
```

Quindi, per compilare ed eseguire il programma, utilizzeremo i seguenti comandi.

```
mpicc -o GameOfLife GameOfLife.c
```

```
mpirun --allow-run-as-root -np 20 GameOfLife 40 10 10
```

Questo comando specifica di eseguire GameOfLife su 20 nodi, di eseguire 40 iterazioni e di utilizzare una matrice composta da 10 righe e da 10 colonne.

1.2.2: Allocazione e generazione matrice iniziale

Recuperate le informazioni passate come parametri, il nodo master (nodo 0) si occuperà di generare la matrice iniziale. Questa matrice verrà allocata dinamicamente e ci preoccuperemo di assicurarci che l'allocazione avvenga in celle contigue di memoria (ciò è necessario per l'utilizzo di MPI).

```c
bool **matrice = alloc_array_bool(righe, colonne); //allocazione dinamica della matrice di boolean
if(my_rank == 0) {
    printf("Numero di iterazioni = %d.\n", iterazioni);
    printf("Numero di righe = %d.\n", righe);
    printf("Numero di colonne = %d.\n\n", colonne);
    srand(my_rank); //inizializzazione random generator
    printf("Stampo la matrice generata.\n");
    for(int c = 0; c < righe; c++) {
        for(int i = 0; i < colonne; i++)
        {
            matrice[c][i] = rand() % 2; //generiamo boolean random
            printf("%d ", matrice[c][i]);
        }
        printf("\n");
    }
}
```

Come è possibile vedere dalla foto precedente, per allocare dinamicamente la matrice in celle contigue di memoria, è stata creata un'apposita funzione "alloc_array_bool" che prende in input il numero di righe e il numero di colonne della matrice. Di seguito vediamo l'implementazione della funzione.

```c
//funzione che ci permette di allocare la matrice dinamicamente e in celle contigue di memoria
//in questo modo possiamo passare le matrici con MPI senza avere problemi
//prende come parametri il numero di righe e il numero di colonne
bool **alloc_array_bool(int righe, int colonne) {
    bool *data = (bool *)malloc(righe*colonne*sizeof(bool));
    bool **array= (bool **)malloc(righe*sizeof(bool*));
    for(int i=0; i<righe; i++)
        array[i] = &(data[colonne*i]);
    return array;
}
```

Arrivati a questo punto, l'esecuzione continua in due possibili modi. Il primo caso prevede l'esecuzione su un unico nodo mentre il secondo caso prevede l'esecuzione su più nodi. Vediamo innanzitutto come avviene l'esecuzione su un unico nodo.

1.2.3: Esecuzione su un unico nodo

Nel caso in cui l'esecuzione avvenga su un unico nodo, andiamo ad allocare la matrice che conterrà il numero di vicini vivi per ogni cellula della matrice generata in precedenza. Fatto ciò, ad ogni iterazione andiamo a calcolare il numero di vicini vivi e successivamente andiamo ad

aggiornare lo stato di ogni cellula (per fare queste due cose utilizzeremo le funzioni “trovaNumeroViciniVivi” e “aggiornaStatoCellule” che spiegheremo in seguito nel dettaglio).

```
if(p == 1) { //esecuzione su un unico nodo
    float time = MPI_Wtime();
    printf("\nSto eseguendo su un unico nodo.\n");
    int **vicini_vivi = (int **)malloc(sizeof(int *)*righe); //allocazione dinamica della matrice contenente il numero di vicini vivi
    for(int i=0; i<righe; i++) {
        vicini_vivi[i] = (int *)malloc(sizeof(int)*colonne); //allocazione dinamica della matrice contenente il numero di vicini vivi
    }
    for(int c = 0; c < iterazioni; c++) {
        printf("\nInizio iterazione %d.\n", c+1);
        trovaNumeroViciniVivi(matrice, righe, colonne, vicini_vivi, 1); //andiamo a ricercare il numero di vicini vivi
        aggiornaStatoCellule(matrice, righe, colonne, vicini_vivi, 1); //andiamo ad aggiornare lo stato delle cellule
        //stampaMatrice(matrice, righe, colonne); //stampiamo per verificare la correttezza
    }
    float etime =  MPI_Wtime() - time;
    stampaMatrice(matrice, righe, colonne); //stampiamo la matrice finale
    printf("\nTime: %f\n\n", etime);
```

Come è possibile vedere, ci occupiamo anche di stampare la matrice finale (tramite la funzione “stampaMatrice” che spiegheremo in seguito nel dettaglio) e di calcolare il tempo di esecuzione di tutto il processo che non comprende il tempo per generare la matrice iniziale (lo stesso vale nei casi in cui operino più di un processo).

1.2.4: Esecuzione su più nodi

Nel caso in cui l’esecuzione avvenga su più nodi, possiamo dividere la soluzione in altri due modi. Un primo caso in cui il numero di processi da utilizzare è maggiore del numero di righe della matrice e un secondo caso in cui il numero di processi da utilizzare è minore o uguale al numero di righe della matrice.

1.2.4: Esecuzione su più nodi (numero di processi maggiore del numero di righe)

Nel caso in cui l’esecuzione debba avvenire con un numero di processi maggiore del numero di righe della matrice, non andremo ad utilizzare tutti i processi ma utilizzeremo un numero di processi pari al numero di righe.

```
if(righe < p) { //se il numero di righe è minore o uguale al numero di processi
    int ranks[righe], new_my_rank;
    int displs[righe]; //indica la posizione dove iniziare ad inviare
    int send_counts_row[righe]; //numero di righe da inviare e da ricevere
    int send_counts[righe]; //numero di elementi da inviare e da ricevere
    MPI_Group orig_group, new_group; //nuovo gruppo
    MPI_Comm new_comm; //nuovo communicator
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group); //estraiamo il gruppo originale
    for(int i = 0; i < righe; i++) {
        ranks[i] = i;
    }
    MPI_Group_incl(orig_group, righe, ranks, &new_group); //creiamo il nuovo gruppo
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm); //creiamo il communicator per il nuovo gruppo
    MPI_Group_rank(new_group, &new_my_rank); //nuovo rank dei processi
```

Innanzitutto, andiamo ad estrarre il gruppo originale (MPI_COMM_WORLD) e partendo da questo gruppo andiamo a creare un nuovo gruppo e un nuovo communicator. Fatto ciò, andremo a recuperare per ogni processo il rank associato al nuovo gruppo creato.

```
if (my_rank < righe) {
    calcolaSendCountsAndDispls(righe, colonne, righe, send_counts_row, send_counts, displs); //calcoliamo il numero
    matrice_rcv = alloc_array_bool(send_counts_row[new_my_rank]+2, colonne); //allocazione dinamica della matrice di
    MPI_Scatterv(&(matrice[0][0]), send_counts, displs, MPI_C_BOOL, &(matrice_rcv[1][0]), send_counts[new_my_rank],
    vicini_vivi = (int **)malloc(sizeof(int *)*send_counts_row[my_rank]); //allocazione dinamica della matrice conte
    for(int i=0; i<send_counts_row[new_my_rank]; i++) {
        vicini_vivi[i] = (int *)malloc(sizeof(int)*colonne); //allocazione dinamica della matrice contenente il nume
    }
```

Creato il nuovo gruppo, andremo ad operare solamente con i processi appartenenti a questo. Quindi, questi processi calcoleranno il numero di righe da inviare, il numero di elementi da inviare e il displacement utilizzando la funzione “calcolaSendCountsAndDispls”. Fatto ciò, allocheremo una matrice di cellule che possa contenere le righe che riceveremo e andremo ad effettuare una Scatterv, ovvero, andremo a suddividere l’array iniziale tra tutti i processi del nuovo gruppo creato. Da notare che il numero di processi che utilizzeremo è uguale al numero di righe. Quindi, in questo caso il numero di righe da inviare sarà sempre una, il numero di elementi da inviare sarà sempre uguale al numero di colonne e il displacement sarà sempre uguale al rank del processo moltiplicato per il numero di colonne.

```
for(int i = 0; i < iterazioni; i++) {
    MPI_Isend(&(matrice_rcv[1][0]), colonne, MPI_C_BOOL, (new_my_rank+righe-1)%righe, 0, new_comm, &request1); /
    MPI_Isend(&(matrice_rcv[send_counts_row[new_my_rank]][0]), colonne, MPI_C_BOOL, (new_my_rank+1)%righe, 1, ne
    MPI_Recv(&(matrice_rcv[send_counts_row[new_my_rank]+1][0]), colonne, MPI_C_BOOL, (new_my_rank+1)%righe, 0, r
    MPI_Recv(&(matrice_rcv[0][0]), colonne, MPI_C_BOOL, (new_my_rank+righe-1)%righe, 1, new_comm, &status); //ri
    trovaNumeroViciniVivi(matrice_rcv, send_counts_row[new_my_rank], colonne, vicini_vivi, 0); //andiamo a ricer
    aggiornaStatoCellule(matrice_rcv, send_counts_row[new_my_rank], colonne, vicini_vivi, 0); //andiamo ad aggio
}
MPI_Gatherv(&(matrice_rcv[1][0]), send_counts[new_my_rank], MPI_C_BOOL, &(matrice[0][0]), send_counts, displs, M
```

Infine, ad ogni iterazioni andremo ad inviare e a ricevere i nuovi valori delle cellule confinanti. L’invio avviene in modo asincrono perché non ci interessa quando il ricevitore riceve i dati inviati mentre la ricezione avviene in maniera sincrona poiché senza il valore delle cellule confinanti non potremmo elaborare i dati. Ricevuti questi valori, andiamo a trovare il numero di vicini vivi per ogni cellula ed andiamo ad aggiornare lo stato delle stesse. Alla fine di tutte le iterazioni, andremo ad effettuare una Gatherv che permetterà al nodo root di ottenere la matrice finale di cellule, la quale potrà essere stampata.

1.2.4: Esecuzione su più nodi (numero di processi minore o uguale al numero di righe)

Nel caso in cui l’esecuzione debba avvenire con un numero di processi minore o uguale al numero di righe della matrice, andremo ad utilizzare tutti i processi.

```
int displs[p]; //indica la posizione dove iniziare ad inviare
int send_counts_row[p]; //numero di righe da inviare e da ricevere
int send_counts[p]; //numero di elementi da inviare e da ricevere
calcolaSendCountsAndDispls(righe, colonne, p, send_counts_row, send_counts, displs); //calcoliamo il numero di elementi d
matrice_rcv = alloc_array_bool(send_counts_row[my_rank]+2, colonne); //allocazione dinamica della matrice di boolean da r
MPI_Scatterv(&(matrice[0][0]), send_counts, displs, MPI_C_BOOL, &(matrice_rcv[1][0]), send_counts[my_rank], MPI_C_BOOL, 0
vicini_vivi = (int **)malloc(sizeof(int *)*send_counts_row[my_rank]); //allocazione dinamica della matrice contenente il
```

In questo caso non andremo a creare il nuovo gruppo ma andremo direttamente a calcolare il numero di righe da inviare, il numero di elementi da inviare e il displacement. Fatto ciò, andremo ad allocare una matrice di cellule che conterrà le cellule che riceveremo e andremo ad effettuare una Scatterv che permetterà di suddividere le righe tra i vari processi.

```
for(int i = 0; i < iterazioni; i++) {
    MPI_Isend(&(matrice_rcv[1][0]), colonne, MPI_C_BOOL, (my_rank+p-1)%p, 0, MPI_COMM_WORLD, &request1); /
    MPI_Isend(&(matrice_rcv[send_counts_row[my_rank]][0]), colonne, MPI_C_BOOL, (my_rank+1)%p, 1, MPI_COMM_
    MPI_Recv(&(matrice_rcv[send_counts_row[my_rank]+1][0]), colonne, MPI_C_BOOL, (my_rank+1)%p, 0, MPI_COMM
    MPI_Recv(&(matrice_rcv[0][0]), colonne, MPI_C_BOOL, (my_rank+p-1)%p, 1, MPI_COMM_WORLD, &status); //ric
    trovaNumeroViciniVivi(matrice_rcv, send_counts_row[my_rank], colonne, vicini_vivi, 0); //andiamo a rice
    aggiornaStatoCellule(matrice_rcv, send_counts_row[my_rank], colonne, vicini_vivi, 0); //andiamo ad aggi
}
MPI_Gatherv(&(matrice_rcv[1][0]), send_counts[my_rank], MPI_C_BOOL, &(matrice[0][0]), send_counts, displs,
```

Fatto ciò, ad ogni iterazione andremo ad inviare e a ricevere i nuovi valori delle cellule confinanti. L'invio avviene in modo asincrono perché non ci interessa quando il ricevitore riceve i dati inviati mentre la ricezione avviene in maniera sincrona poiché senza il valore delle cellule confinanti non potremmo elaborare i dati. Ricevuti questi valori, andiamo a trovare il numero di vicini vivi per ogni cellula ed andiamo ad aggiornare lo stato delle stesse. Alla fine di tutte le iterazioni, andremo ad effettuare una Gatherv che permetterà al nodo root di ottenere la matrice finale di cellule, la quale potrà essere stampata.

1.2.5: Funzione TrovaNumeroViciniVivi

La funzione "TrovaNumeroViciniVivi" si occupa di determinare per ogni elemento della matrice di cellule il numero di vicini vivi. Questa funzione prende come parametri la matrice di cellule, il numero di righe di cui è composta la matrice di cellule, il numero di colonne di cui è composta la matrice di cellule, la matrice dove andremo a salvare il numero di vicini vivi per ogni cellula e un flag che serve a determinare se l'esecuzione sta avvenendo su un unico nodo (valore 1) o su più nodi (valore 0). Il parametro flag è fondamentale poiché se l'esecuzione avviene su un unico nodo, allora bisogna calcolare i vicini di tutte le righe, mentre nel caso in cui l'esecuzione avviene su più nodi, non bisogna calcolare i vicini della prima e dell'ultima riga della matrice perché sarà compito di altri processi.

```
//funzione che ci permette di trovare il numero di vicini vivi
//prende come argomento la matrice di cellule, il numero di righe della matrice,
//il numero di colonne della matrice e la matrice dove salveremo il numero di vicini vivi
//prende anche un boolean che indica se calcolare su tutte le righe (1)
//o se non contare la prima e l'ultima riga (0 in caso di esecuzione su più nodi)
//in particolare, flag sarà uguale a 1 se l'esecuzione sarà su un unico nodo, altrimenti sarà uguale a 0
//non restituisce alcun valore e per ogni elemento della matrice di cellule, andiamo a vedere quanti tra gli 8 vicini sono vivi
//infine aggiorniamo la matrice contentente il numero di vicini vivi per ogni elemento della matrice di cellule
//per verificare la correttezza ci facciamo stampare il numero sullo standard di output
void trovaNumeroViciniVivi(bool **matrice, int righe, int colonne, int **vicini_vivi, bool flag) {
```

1.2.6: Funzione aggiornaStatoCellule

La funzione "aggiornaStatoCellule" si occupa di aggiornare lo stato di ogni cellula della matrice. Questa funzione prende come parametri la matrice di cellule, il numero di righe di cui è composta la matrice di cellule, il numero di colonne di cui è composta la matrice di cellule, la matrice contenente il numero di vicini vivi per ogni cellula e un flag che serve a determinare se l'esecuzione sta avvenendo su un unico nodo (valore 1) o su più nodi (valore 0). Il parametro flag è fondamentale poiché se l'esecuzione avviene su un unico nodo, allora bisogna determinare il nuovo stato di ogni cellula della matrice, mentre nel caso in cui l'esecuzione avviene su più nodi, non bisogna calcolare lo stato della prima e dell'ultima riga della matrice perché sarà compito di altri processi.

```
//funzione che ci permette di aggiornare lo stato delle cellule
//prende come argomento la matrice di cellule, il numero di righe della matrice,
// il numero di colonne della matrice e la matrice che contiene il numero di vicini vivi
//prende anche un boolean che indica se calcolare su tutte le righe (1)
//o se non contare la prima e l'ultima riga (0 in caso di esecuzione su più nodi)
//in particolare, flag sarà uguale a 1 se l'esecuzione sarà su un unico nodo, altrimenti sarà uguale a 0
//non restituisce alcun valore e per ogni elemento della matrice di cellule, andiamo a vedere il nuovo stato
//se una cellula è viva ed ha meno di due vicini vivi --> muore
//se una cellula è viva ed ha più di tre vicini vivi --> muore
//se una cellula è viva ed ha due o tre vivini vivi --> continua a vivere --> non abbiamo bisogno
//di implementare questo caso perchè in realtà non causa alcun aggiornamento
//se una cellula è morta ed ha esettamente tre vicini vivi --> torna a vivere
//per verificare la correttezza facciamo stampare
void aggiornaStatoCellule(bool **matrice, int righe, int colonne, int **vicini_vivi, bool flag) {
```

1.2.7: Funzione stampaMatrice

La funzione “stampaMatrice” permette di stampare la matrice di cellule. Questa funzione prende come parametri la matrice di cellule da stampare, il numero di righe di cui è composta la matrice di cellule e il numero di colonne di cui è composta la matrice di cellule.

```
//funzione che ci permette di stampare la matrice di cellule
//prende come parametri la matrice di cellule da stampare, il numero di righe e il numero di colonne
void stampaMatrice(bool **matrice, int righe, int colonne) {
```

1.2.8: Funzione alloc_array_bool

La funzione “alloc_array_bool” permette di allocare una matrice dinamicamente e in celle contigue di memoria. Ciò ci permette di passare una matrice dinamica con MPI senza avere problemi. Questa matrice prende come parametri il numero di righe e il numero di colonne da cui deve essere composta la matrice da allocare. Inoltre, viene restituita la matrice allocata.

```
//funzione che ci permette di allocare la matrice dinamicamente e in celle contigue di memoria
//in questo modo possiamo passare le matrici con MPI senza avere problemi
//prende come parametri il numero di righe e il numero di colonne
bool **alloc_array_bool(int righe, int colonne) {
```

1.2.9: Funzione calcolaSendCountsAndDispls

La funzione “calcolaSendCountsAndDispls” permette di calcolare il numero di righe da inviare, il numero di elementi da inviare e il displacement per ogni processo. Questa funzione prende come parametri il numero di righe della matrice originale, il numero di colonne della matrice originale, il numero di processi che devono cooperare tra loro, un puntatore ad un array che conterrà il numero di righe da inviare per ogni nodo, un puntatore ad un array che conterrà il numero di elementi da inviare per ogni nodo e un puntatore ad un array che conterrà il displacement per ogni nodo.

```
//funzione che calcola il numero di elementi da inviare, il numero di righe e il displacement
//prende come parametri il numero di righe della matrice originale,
//il numero di colonne della matrice originale e il numero di processi
//inoltre, prende le variabili dove salvare il numero di elementi da inviare,
//il numero di righe da inviare e i displacement per i vari processi
void calcolaSendCountsAndDispls(int righe, int colonne, int p, int *send_counts_row, int *send_counts, int *displs) {
```

1.3: Benchmarking

Il benchmarking dell’implementazione è stato effettuato su un cluster di quattro macchine “t2.2xlarge” per un totale di 32 vCPUs. Sono stati effettuati un test per valutare la scalabilità forte e un test per valutare la scalabilità debole. I test effettuati sono i seguenti:

- matrice 5000*5000 con 40 ripetizioni su 1, 2, 4, 8, 12, 16, 20, 24, 28, 32 vCPUs (scalabilità forte);
- matrice di 5000 colonne con 40 ripetizioni su 1, 2, 4, 8, 12, 16, 20, 24, 28, 32 vCPUs. Andremo ad utilizzare un numero di righe variabile determinato dal numero di vCPUs moltiplicato per 1000 (scalabilità debole).

1.3.1: Scalabilità forte

Per testare la scalabilità forte è stato effettuato un test utilizzando una matrice con 5000 righe e 5000 colonne. Inoltre, sono state effettuate 40 ripetizioni ed il tutto è stato verificato su 1, 2, 4, 8, 12, 16, 20, 24, 28, 32 vCPUs. Oltre a calcolare i tempi di esecuzione, è stata calcolata anche l'efficienza, la quale è data dai tempi di esecuzione del programma tramite un unico nodo diviso per il prodotto tra il numero di processi e il tempo di esecuzione utilizzando questo numero di processi. Il tutto viene moltiplicato per 100:

- $Ep = T_1 / (P * T_P) * 100$.

L'efficienza viene espressa in percentuale mentre il tempo in secondi.

vCPUs	1	2	4	8	12	16	20	24	28	32
Tempo	58.41	30.51	15.38	7.74	5.33	4.11	3.35	2.90	2.58	2.34
Efficienza (%)	100	95.72	94.94	94.33	91.32	88,82	87.17	83.92	80.85	78.00

1.3.2: Scalabilità debole

Per testare la scalabilità debole è stato effettuato un test utilizzando una matrice con 5000 colonne ed un numero di righe pari al numero di processi moltiplicati per 1000. Inoltre, sono state effettuate 40 ripetizioni ed il tutto è stato verificato su 1, 2, 4, 8, 12, 16, 20, 24, 28, 32 vCPUs. Oltre a calcolare i tempi di esecuzione, è stata calcolata anche l'efficienza, la quale è data dai tempi di esecuzione del programma tramite un unico nodo diviso per il tempo di esecuzione utilizzando un numero di processi pari a P. Il tutto viene moltiplicato per 100:

- $Ep = (T_1 / T_P) * 100$.

vCPUs	1	2	4	8	12	16	20	24	28	32
N. righe	1000	2000	4000	8000	12000	16000	20000	24000	28000	32000
Tempo	12.28	12.30	12.35	12.33	12.67	12.99	13.29	13.83	13.98	14.31
Efficienza (%)	100	99.83	99.43	99.59	96.92	94.53	92.40	88.79	87.83	85.81

1.3.3: Risultati

I risultati mostrano che attraverso l'utilizzo delle computazione parallela avviene un importante incremento delle performance per quanto riguarda i tempi di esecuzione. Ovviamente, all'aumentare del numero di processori il miglioramento tende a diminuire leggermente per via dell'overhead dovuto alla maggiore comunicazione. Nonostante ciò, fino a 32 nodi possiamo notare che i risultati sono comunque ottimi, come è possibile vedere anche tramite il calcolo dell'efficienza. Durante il testing ci siamo limitati a soli 32 nodi ma ci aspettiamo che con un ampio incremento del numero di nodi, il miglioramento tenda a scomparire.

1.4: Correttezza

Per valutare la correttezza dell'algoritmo abbiamo verificato che, utilizzando una matrice costante con un numero di core variabile, la matrice finale restituita sia sempre la stessa. Di seguito vediamo gli screen di varie esecuzioni. Innanzitutto, vediamo cosa succede quando eseguiamo su un unico nodo.

```
root@e460140359dc:/home/progetto# mpirun --allow-run-as-root -np 1 GameOfLife 5 8 6
Numero di iterazioni = 5.
Numero di righe = 8.
Numero di colonne = 6.

Stampo la matrice generata.
1 0 1 1 1 1
0 0 1 1 0 1
0 1 1 0 0 0
0 0 1 0 1 1
0 0 0 1 1 1
1 0 0 0 1 1
1 0 1 0 1 1
1 1 0 1 0 0

Sto eseguendo su un unico nodo.

Stampo la matrice finale.

0 0 0 0 0 0
0 0 0 0 1 0
0 0 1 0 0 1
0 0 1 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Ora vediamo cosa succede quando eseguiamo su un numero di nodi maggiore del numero di righe.

```
root@e460140359dc:/home/progetto# mpirun --allow-run-as-root -np 12 GameOfLife 5 8 6
Numero di iterazioni = 5.
Numero di righe = 8.
Numero di colonne = 6.

Stampo la matrice generata.
1 0 1 1 1 1
0 0 1 1 0 1
0 1 1 0 0 0
0 0 1 0 1 1
0 0 0 1 1 1
1 0 0 0 1 1
1 0 1 0 1 1
1 1 0 1 0 0
Sto eseguendo su più nodi.

0 0 0 0 0 0
0 0 0 0 1 0
0 0 1 0 0 1
0 0 1 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Ora vediamo cosa succede quando eseguiamo su un numero di nodi minore del numero di righe.

```
root@e460140359dc:/home/progetto# mpirun --allow-run-as-root -np 5 GameOfLife 5 8 6
Numero di iterazioni = 5.
Numero di righe = 8.
Numero di colonne = 6.

Stampo la matrice generata.
1 0 1 1 1 1
0 0 1 1 0 1
0 1 1 0 0 0
0 0 1 0 1 1
0 0 0 1 1 1
1 0 0 0 1 1
1 0 1 0 1 1
1 1 0 1 0 0
Sto eseguendo su più nodi.

0 0 0 0 0 0
0 0 0 0 1 0
0 0 1 0 0 1
0 0 1 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Infine, vediamo cosa succede quando eseguiamo su un numero di nodi pari al numero di righe.

```
root@e460140359dc:/home/progetto# mpirun --allow-run-as-root -np 8 GameOfLife 5 8 6
Numero di iterazioni = 5.
Numero di righe = 8.
Numero di colonne = 6.

Stampo la matrice generata.
1 0 1 1 1 1
0 0 1 1 0 1
0 1 1 0 0 0
0 0 1 0 1 1
0 0 0 1 1 1
1 0 0 0 1 1
1 0 1 0 1 1
1 1 0 1 0 0
Sto eseguendo su più nodi.

0 0 0 0 0 0
0 0 0 0 1 0
0 0 1 0 0 1
0 0 1 0 1 0
0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Come è possibile vedere, la soluzione proposta opera sempre allo stesso modo indipendentemente dal numero di processori utilizzati. Da ciò ne deduciamo che la soluzione proposta svolga perfettamente il compito richiesto.

1.5: Conclusioni

L'implementazione proposta realizza una buona soluzione per Game of Life sfruttando sia la programmazione sequenziale sia la programmazione parallela. Tramite i risultati, è possibile notare che parallelizzando il problema si ottengono miglioramenti importanti per quanto riguarda la computazione e che in questa implementazione il tempo di overhead non ha un peso importante. Ciò è dovuto al fatto che si è riuscito a ridurre al minimo il numero di comunicazioni tra i processi. Infatti, per ogni iterazione ogni processo ha bisogno esclusivamente di ricevere due righe da due processi differenti e di inviare due righe a due processi differenti. Inoltre, l'invio delle righe avviene in modo asincrono e ciò evita ritardi dovuti alla ricezione da parte degli altri nodi.