UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

RELAZIONE - GAME OF LIFE

Professori:

Chiar.mo Prof. Vittorio Scarano
Chiar.mo. Prof. Carmine Spagnuolo

Studente:

Andrea Massaro
Matricola: 0522500795

Anno Accademico 2020/2021

Sommario

ABSTRACT ---------- **2**

CAPITOLO 1 ---------- **3**

INTRODUZIONE ---------- 4

SOLUZIONE PROPOSTA ---------- 5

CAPITOLO 2 ---------- **6**

LE VARIABILI ---------- 7

FUNZIONI MPI UTILIZZATE ---------- 8

ANALISI DEL NUMERO DI PROCESSORI ---------- 9

LA FINTA MATRICE DEL MONDO DI GIOCO ---------- 10

LA DIVISIONE DELLA MATRICE ---------- 12

VIVO O MORTO? ---------- 14

CAPITOLO 3 ---------- **18**

COME SONO STATE CONDOTTE LE MISURAZIONI ---------- 19

SCALABILITÀ DEBOLE ---------- 20

SCALABILITÀ FORTE ---------- 22

CAPITOLO 4 ---------- **26**

CONCLUSIONI ---------- 26

ISTRUZIONI PER UNA CORRETTA ESECUZIONE ---------- 27

ABSTRACT

Il seguente documento ha l'obiettivo di fornire una descrizione dettagliata sullo svolgimento della parte pratica (laboratorio) dell'esame universitario "Programmazione Concorrente, Parallela e su Cloud". Il progetto realizzato è basato sulla creazione di un programma scritto nel linguaggio di programmazione C con l'obiettivo di implementare le varie tecniche di parallelizzazione studiate durante la prima parte del corso, sfruttando la libreria "mpi.h" fornita per il protocollo di comunicazione Message Passing Interface (MPI).

Volendo essere più precisi, si andrà a:

1. Fornire una descrizione formale del problema affrontato
2. Discussione della soluzione proposta
3. Analisi della soluzione implementata
4. Analisi delle prestazioni ottenute eseguendo l'algoritmo sull'infrastruttura EC2, su macchine di categoria t2.large
5. Conclusioni

Risultato MD5: 3cb62d591b6ca5fc3ada397b98795165

Progetto 1 - Instance Type 2

Game of Life – t2.large

CAPITOLO 1

INTRODUZIONE

INTRODUZIONE

Game of Life, proprio come riporta Wikipedia, è un automa cellulare sviluppato dal matematico inglese John Conway sul finire degli anni Sessanta. Il suo scopo è quello di mostrare come comportamenti simili alla vita possano emergere da regole semplici e interazioni a molti corpi, principio che è alla base dell'eco-biologia, la quale si rifà anche alla teoria della complessità.

Si tratta di un gioco senza giocatori ovvero, senza che vi sia un giocatore a fornire input continui, continuerà in totale autonomia. Il gioco si svolge su una griglia, o matrice, di dimensioni MxN di forma toroidale. Ogni cella ha otto vicini, ovvero tutte i quadrati ad essa adiacente.

Ogni cella può trovarsi in soli due stati:

1. viva
2. morta

Lo stato della griglia evolve ciclo dopo ciclo e, seconde regole ben definite, le celle potranno morire o iniziare a vivere.

Soluzione proposta

Il mondo di gioco è salvato in un array monodimensionale e trattato come fosse una matrice di forma toroidale. La matrice può essere di qualsiasi dimensione NxM, la durata del gioco viene definita dal giocatore per mezzo degli argomenti da passare per l'esecuzione del programma ed i valori all'interno della matrice sono di origine random il cui seme viene fornito dall'utente. Questa implementazione, inoltre, funziona con qualsiasi numero di processori forniti dal giocatore, la gestione di un eventuale sovrannumero di essi sarà discussa più avanti. La soluzione proposta cerca di minimizzare la sequenzialità e sfruttare l'enorme potenzialità della computazione parallela. Tutti i processori, incluso il master, partecipano alla computazione.

L'algoritmo è così suddiviso:

1. Inizializzazione delle variabili necessarie
2. Analisi del numero di processori e della grandezza della matrice
3. Ottimizzazione del numero di processori
4. Allocazione dinamica di array secondo le grandezze definite dall'utente
5. Creazione del mondo di gioco secondo un seme definito dall'utente
6. Suddivisione del mondo di gioco fra tutti i processori coinvolti
7. Calcolo della nuova generazione
8. Output del nuovo mondo di gioco

CAPITOLO 2

Descrizione della soluzione

Le variabili

Prima di descrivere la soluzione, si andranno a descrivere le variabili più importanti:

1. Matrix, array che memorizza il mondo di gioco. La scelta di usare l'array al posto di una matrice viene discussa più avanti
2. Before: array che memorizza la ghost line che precede la prima riga ricevuta
3. After: array che memorizza la ghost line che segue l'ultima riga ricevuta
4. ReciverCounts: array di interi in cui memorizzare il numero di elementi da inviare all'i-esimo processore
5. ReciverDispacement: array di interi in cui memorizzare il punto di partenza di ogni segmento da inviare ai processori coinvolti

Funzioni MPI utilizzate

Ci si è avvalsi delle seguenti funzioni MPI:

1. MPI_Comm_group();
2. MPI_gruoup_range_excl();
3. MPI_Comm_create();
4. MPI_Barrier();
5. MPI_Scatterv();
6. MPI_Isend();
7. MPI_Irecv();
8. MPI_Waitall();
9. MPI_Gatherv();

Analisi del numero di processori

L'algoritmo proposto, come vedremo successivamente, divide le righe in base al numero di processori a disposizione. Nel caso, però, il numero di righe sia inferiore al numero di core fornito dall'utente si avrebbero comportamenti anomali da parte del programma.

Per ovviare a questo problema, si è deciso di adottare la seguente soluzione:

```
MPI_Group new_group;
MPI_Group world_group;
MPI_Comm MPI_NEW_COMM_WORLD;
```

```
//CREAZIONE DI UN NUOVO COMUNICATORE PER EVITARE L'INIZIALIZZAZIONE DI PROCESSI IN ECCESSO DA PARTE DELL'UTENTE
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
if (numberOfProcesses > rows)
{
    int ranges[][3] = {rows, numberOfProcesses - 1, 1};
    MPI_Group_range_excl(world_group, 1, ranges, &new_group);
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &MPI_NEW_COMM_WORLD);
}
else
{
    MPI_Comm_create(MPI_COMM_WORLD, world_group, &MPI_NEW_COMM_WORLD);
}
if (MPI_NEW_COMM_WORLD == MPI_COMM_NULL)
{
    // Bye bye cruel world
    MPI_Finalize();
    exit(0);
}
```

1. Creazione di un nuovo gruppo di processori, world_group, ed esclusione dei processi in eccesso
2. Creazione di un nuovo comunicatore, MPI_NEW_COMM_WORLD
3. Eliminazione dei processori in eccesso

La finta matrice del mondo di gioco

Per l'implementazione del mondo di gioco si potrebbe procedere in due modi:

1. Creazione di una matrice NxM, dove N ed M sono rispettivamente le righe e le colonne definite dall'utente
2. Creazione di un singolo array di dimensione $N * M$

Dopo un'attenta analisi, la seconda soluzione è parsa la più conveniente per i seguenti motivi:

1. L'array monodimensionale viene caricato interamente in memoria cache, o al massimo la riempie interamente nel caso il numero di byte occupati sia superiore a quello messo a disposizione dalla macchina t2.large. Cosa diversa avviene quando si utilizza una matrice perché, essendo C un linguaggio row based, viene caricata solo la riga a cui si sta puntando.
2. Diretta conseguenza del punto 1, il tempo di accesso alle informazioni è nettamente più rapido non dovendo caricare una nuova riga ogni volta che si giunge all'elemento finale di una linea.

L'algoritmo risultante da tale scelta avrà sicuramente una complessità di lettura maggiore, ma i guadagni in termini di performance ottenuti giustificano pienamente tale scelta.

Il tipo di dato utilizzato è il Bool, definito nella libreria stdbool.h. La scelta deriva dalle seguenti motivazioni:

1. Il tipo *Bool* ed il tipo *char* hanno la medesima grandezza, ovvero 1 byte. Il tipo *int* ha grandezza 4 byte ed è stato scartato a priori
2. Non effettuando conversioni di tipo, cosa che sarebbe stata necessaria nel caso in cui ci si fosse affidati al char, si rende l'algoritmo più leggibile e meno complesso da scrivere.

```
//INIZIALIZZAZIONE E STAMPA DEL MONDO DI GIOCO
if (myRank == 0)
{
    srand(seed);
    matrix = (bool *)calloc((rows * cols), sizeof(bool));
    printf("Seed: %d\nGeneration 0 auto-generated:\n", seed);
    fflush(stdout);
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            matrix[i * cols + j] = rand() % 2;
            printf("%d\t", matrix[i * cols + j]);
            fflush(stdout);
        }
        printf("\n");
        fflush(stdout);
    }
    printf("\n");
    fflush(stdout);
}
```

Com'è possibile apprezzare nell'immagine sopra, il processore con rank 0 si occupa dell'inizializzazione della matrice e del suo riempimento.

La divisione della matrice

La suddivisione della matrice si avvale della funzione MPI_Scatterv. I vantaggi di tale soluzione sono:

1. La possibilità di dividere in modo arbitrario l'array di gioco, ovvero è possibile dare ad ogni processore una quantità diversa di dati da analizzare
2. Elevate performance e chiara comprensione del codice poiché si evita di chiamare innumerevoli volte le funzioni MPI_Send ed MPI_Recv (sincrone), o MPI_Isend ed MPI_Irecv (asincrone)

```
surplus = rows % numberOfProcesses;
```

```
//DEFINIZIONE DEL NUMERO DI ELEMENTI DA DISTRIBUIRE AI PROCESSORI E DA QUALE ELEMENTO PARTIRE ALL'INTERNO DELLA "MATRICE"
for (int i = 0; i < numberOfProcesses; i++)
{
    reciverCounts[i] = (surplus - i > 0) ? ((cols * (rows / numberOfProcesses)) + cols) : (cols * (rows / numberOfProcesses));
    beforeAfterCounts[i] = cols;
    reciverDisplacement[i] = (i == 0) ? 0 : (reciverCounts[i - 1] + reciverDisplacement[i - 1]);
    beforeDisplacement[i] = (i == 0) ? ((rows * cols) - cols) : reciverDisplacement[i] - cols;
    afterDisplacement[i] = (i == numberOfProcesses - 1) ? 0 : reciverCounts[i] + reciverDisplacement[i];
}
```

```
//DISTRIBUZIONE DEL MONDO DI GIOCO
MPI_Scatterv(matrix, reciverCounts, reciverDisplacement, MPI_C_BOOL, reciver, reciverCounts[myRank], MPI_C_BOOL, 0, MPI_NEW_COMM_WORLD);
```

La soluzione implementata esegue le seguenti operazioni:

1. Calcolo il surplus nel caso in cui il numero di righe non sia un multiplo del numero dei processori disponibili

2. Calcolo del numero di elementi da assegnare ad ogni processore. Se surplus – i, dove i è l'i-esimo processo, dovesse risultare maggiore di 0, questo riceverà una riga in più
3. BeforeAfterCounts dell'i-esimo processo sarà sempre pari al numero di colonne, questo perchè rappresenta il numero di elementi che compone sia la riga ghost successiva che quella precedente
4. ReciverDisplacement è utilizzato dalla MPI_Scattterv per sapere da quale posizione della "matrice" iniziare ad inviare dati all'i-esimo processo
5. BeforeDisplacement ed afterDisplacement hanno la stessa finalità di reciverDisplacement ma applicata alle righe ghost, ovvero l'ultima riga assegnata al processore precedente e la prima assegnata a quello successivo

Vivo o morto?

Game of life, come descritto in precedenza, si basa su quattro semplici regole:

1. Se una cella è viva ed ha meno di due vicini vivi, muore per declino demografico
2. Se una cella ha due o tre vicini vivi continua a vivere
3. Se una cella viva ha quattro o più vicini vivi, muore per sovrappopolamento
4. Se una cella morta ha esattamente tre vicini vivi, questa comincia a vivere per replicazione

Questa sezione dell'algoritmo può essere divisa in due parti: una che non tiene conto del numero di righe ricevute, l'altra che ottimizza il calcolo parallelo.

```
//PREPARAZIONE DELLA GHOST LINE PER IL PROCESSO PRECEDENTE
for (int i = 0; i < cols; i++)
{
    toSendAfter[i] = reciver[i];
}

sendTo = (myRank == 0) ? (numberOfProcesses - 1) : (myRank - 1);
MPI_Isend(toSendAfter, cols, MPI_C_BOOL, sendTo, 0, MPI_NEW_COMM_WORLD, &request);

reciveFrom = (myRank == numberOfProcesses - 1) ? 0 : (myRank + 1);
MPI_Irecv(after, cols, MPI_C_BOOL, reciveFrom, 0, MPI_NEW_COMM_WORLD, &beforeAfterIrecv[0]);

//PREPARAZIONE DELLA GHOST LINE PER IL PROCESSO SUCCESSIVO
if (reciverCounts[myRank] / cols == 1)
{
    for (int i = 0; i < cols; i++)
    {
        toSendBefore[i] = reciver[i];
    }
    sendTo = (myRank == numberOfProcesses - 1) ? 0 : myRank + 1;
    MPI_Isend(toSendBefore, cols, MPI_C_BOOL, sendTo, 0, MPI_NEW_COMM_WORLD, &request);
    reciveFrom = (myRank == 0) ? numberOfProcesses - 1 : myRank - 1;
    MPI_Irecv(before, cols, MPI_C_BOOL, reciveFrom, 0, MPI_NEW_COMM_WORLD, &beforeAfterIrecv[1]);
}
else
{
    int j = 0;
    for (int i = (reciverCounts[myRank] - cols); i < reciverCounts[myRank]; i++)
    {
        toSendBefore[j] = reciver[i];
        j++;
    }
    sendTo = (myRank == numberOfProcesses - 1) ? 0 : myRank + 1;
    MPI_Isend(toSendBefore, cols, MPI_C_BOOL, sendTo, 0, MPI_NEW_COMM_WORLD, &request);
    reciveFrom = (myRank == 0) ? numberOfProcesses - 1 : myRank - 1;
    MPI_Irecv(before, cols, MPI_C_BOOL, reciveFrom, 0, MPI_NEW_COMM_WORLD, &beforeAfterIrecv[1]);
}
```

Ora passiamo ad analizzare la parte ottimizzata per il calcolo parallelo:

```
// SE LE RIGHE RICEVUTE SONO MINORI DI DUE
if (reciverCounts[myRank] / cols <= 2)
{
    //ATTENDO LE GHOST LINES
    MPI_Waitall(2, beforeAfterIrecv, status);

    //CALCOLO IL NUOVO MINI-MONDO DI GIOCO
    makePrivateMatrix(reciverCounts, myRank, cols, reciver, before, after, sender, 0, reciverCounts[myRank]);
}

// SE LE RIGHE RICEVUTE SONO MAGGIORI DI DUE
else
{
    //CALCOLO IL NUOVO MINI-MONDO DI GIOCO ESCLUDENDO LA PRIMA ED ULTIMA LINEA
    makePrivateMatrix(reciverCounts, myRank, cols, reciver, before, after, sender, 1, (reciverCounts[myRank] / cols) - 1);

    //ATTENDO LE GHOST LINES
    MPI_Waitall(2, beforeAfterIrecv, status);

    //PRIMA LINEQ
    makePrivateMatrix(reciverCounts, myRank, cols, reciver, before, after, sender, 0, 1);

    //ULTIMA RIGA
    makePrivateMatrix(reciverCounts, myRank, cols, reciver, before, after, sender, (reciverCounts[myRank] / cols) - 1, reciverCounts[myRank] / cols);
}
```

Nel caso in cui le righe ricevute siano minori o uguali a due, è necessario attendere l'arrivo delle righe ghost degli altri processori al fine di poter calcolare il valore delle celle. Per fare ciò ci si è avvalsi della funzione MPI_Waitall.

Invece, se le righe ricevute dovessero essere di numero maggiore, vengono prima calcolate quelle centrali, si aspetta che le ghost lines siano state memorizzate, ed infine si calcolano i valori della prima e dell'ultima riga. In questo modo si prova a ridurre al minimo il tempo sprecato per la comunicazione, anche se, come vedremo successivamente, l'adozione della struttura array al posto della matrice rende il programma così efficiente che la computazione termina molto prima della comunicazione, rendendo quest'ultima un collo di bottiglia notevole.

```
void makePrivateMatrix(int *reciverCounts, int myRank, int cols, bool *reciver, bool *before, bool *after, bool *sender, int startFrom, int arriveTo)
{
    int block[9];
    int livingCells = 0;
    for (int i = startFrom; i < arriveTo / cols; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            //focus = 0, left = 1, right = 2, up = 3, down = 4, lUpCorner = 5, lDownCorner = 6, rUpCorner = 7, rDownCorner = 8;
            block[0] = reciver[i * cols + j];
            block[1] = (j == 0) ? (reciver[((i * cols + cols) - 1)]) : (reciver[((i * cols + j) - 1)]);
            block[2] = (j == cols - 1) ? (reciver[(i * cols)]) : (reciver[((i * cols + j) + 1)]);
            block[3] = (i == 0) ? (before[j]) : (reciver[(((i - 1) * cols) + j)]);
            block[4] = (i == ((reciverCounts[myRank] / cols) - 1)) ? (after[j]) : (reciver[(((i + 1) * cols) + j)]);

            if (i == 0)
            {
                block[5] = (j == 0) ? before[cols - 1] : before[j - 1];
            }
            else if (i > 0)
            {
                block[5] = (j == 0) ? reciver[(((((i - 1) * cols) + cols) - 1)] : reciver[((((i - 1) * cols) + j) - 1)];
            }

            if (i == ((reciverCounts[myRank] / cols) - 1))
            {
                block[6] = (j == 0) ? after[cols - 1] : after[j - 1];
            }
            else if (i < ((reciverCounts[myRank] / cols) - 1))
            {
                block[6] = (j == 0) ? reciver[(((((i + 1) * cols) + cols) - 1)] : reciver[((((i + 1) * cols) + j) - 1)];
            }

            if (i == 0)
            {
                block[7] = (j == cols - 1) ? before[0] : before[j + 1];
            }
            else if (i > 0)
            {
                block[7] = (j == cols - 1) ? reciver[((i - 1) * cols)] : reciver[((((i - 1) * cols) + j) + 1)];
            }

            if (i == ((reciverCounts[myRank] / cols) - 1))
            {
                block[8] = (j == cols - 1) ? after[0] : after[j + 1];
            }
            else if (i < ((reciverCounts[myRank] / cols) - 1))
            {
                block[8] = (j == cols - 1) ? reciver[(i + 1) * cols] : reciver[((((i + 1) * cols) + j) + 1)];
            }
```

```
//ANALIZZO IL NUMERO DI CELLE VIVE ED APPLICO LE REGOLE DI VIVO-MORTO
for (int k = 1; k < 9; k++)
{
    livingCells = livingCells + block[k];
    if (block[0] == 1 && livingCells > 3)
    {
        sender[i * cols + j] = 0;
        break;
    }
}

if (block[0] == 0 && livingCells == 3)
{
    sender[i * cols + j] = 1;
}
else if (block[0] == 1 && (livingCells == 2 || livingCells == 3))
{
    sender[i * cols + j] = 1;
}
else if (block[0] == 1 && livingCells < 2)
{
    sender[i * cols + j] = 0;
}
livingCells = 0;
```

Nell'immagine sopra andiamo ad analizzare il modo in cui si modifica lo stato di ogni cella presa in esame:

1. Si esegue un ciclo su tutte le celle di block. Se la cella d'interesse è vicva ed il numero di livingCells è maggiore di tre, questa viene dichiarata morta e si esce dal ciclo
2. Se la cella è morta e sono presenti tre vicini, lo stato passa a viva
3. Se la cella è viva ed è circondata da due o tre celle vive, lo stato non muta
4. Se la cella è viva ma non ha abbastanza vicini vivi, lo stato cambia

CAPITOLO 3

Misurazioni

Come sono state condotte le misurazioni

Sono state istanziate 8 VM t2.large, ognuna dotata di 2 vCPUs, quindi si poteva godere di 16 vCPUs in totale.

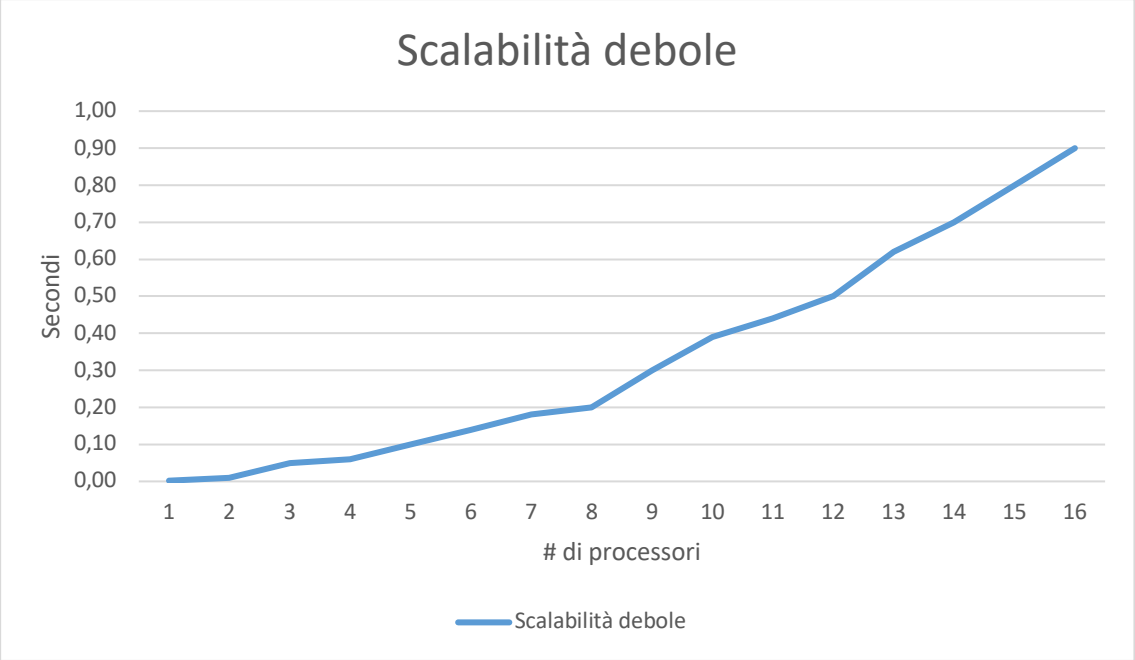
Il mondo di gioco è una matrice quadrata NxN. Si è deciso di optare per questa scelta, benché l'algoritmo possa funzionare anche con matrici rettangolari, per semplicità nel calcolare il numero di righe/colonne da assegnare ad ogni fase di test.

Scalabilità debole

Per mezzo della scalabilità debole si cerca di capire se il tempo per la computazione cresca in modo lineare alla crescita proporzionale del carico di lavoro.

Scalabilità debole con singola MPI_Gaterv alla fine di tutte le simulazioni

Iterazioni=25		
vCPUs	**Tempo**	**# righe/colonne**
1	0,002	312
2	0,009	624
3	0,05	936
4	0,06	1248
5	0,1	1560
6	0,14	1872
7	0,18	2184
8	0,2	2496
9	0,3	2808
10	0,39	3120
11	0,44	3432
12	0,5	3744
13	0,62	4056
14	0,7	4368
15	0,8	4680
16	0,9	4992



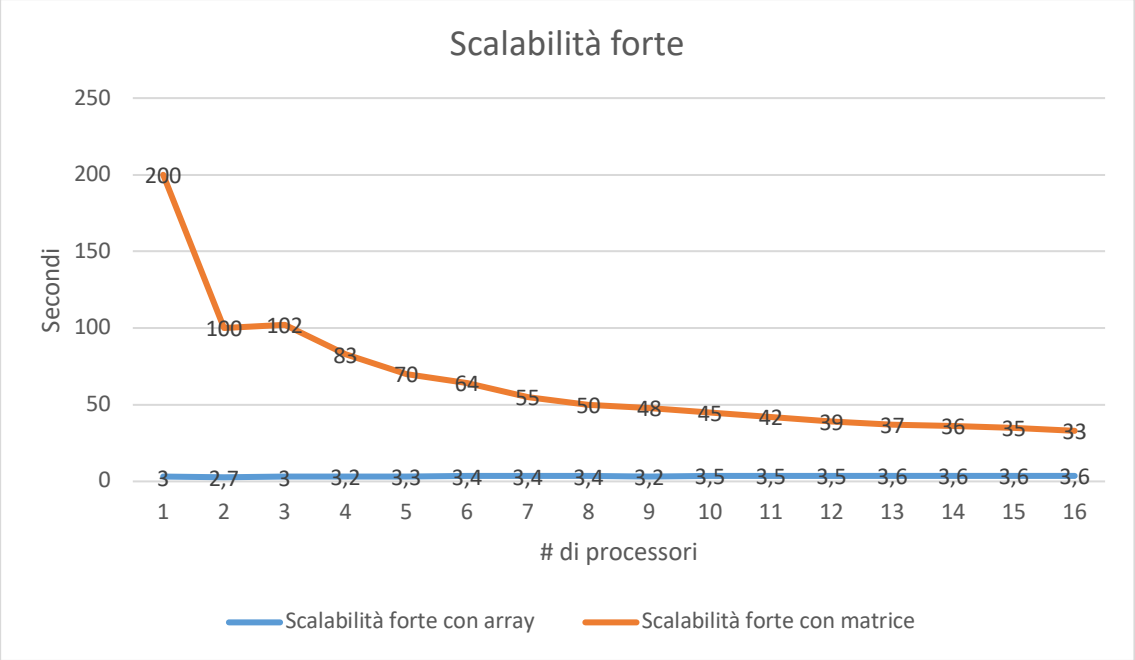
Com’è possibile constatare dal grafico, la crescita del tempo è pressoché lineare.

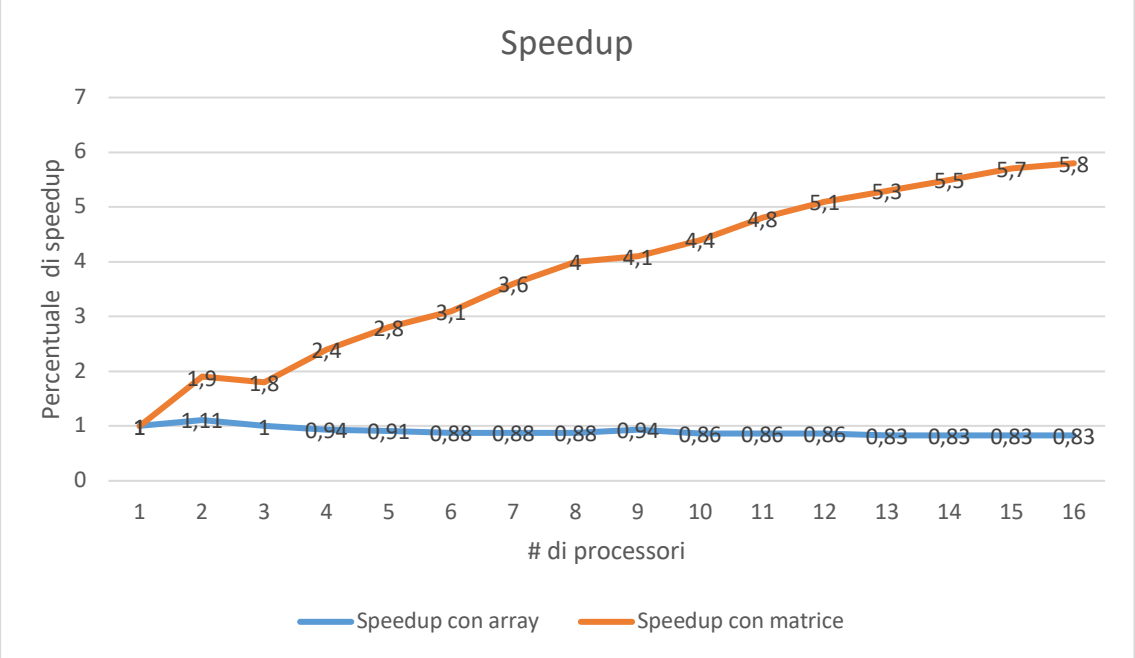
Scalabilità forte

Per mezzo della scalabilità forte si cerca di capire quanto l'algoritmo migliori l'efficienza all'aumentare del numero dei processori, mantenendo inalterato, nel nostro caso, il numero di righe/colonne del mondo di gioco.

Scalabilità forte con singola MPI_Gaterv alla fine di tutte le simulazioni

Iterazioni=25			
vCPUs	**Tempo**	**# righe/colonne**	**Speedup**
1	3,0	10.000	1,0
2	2,7	10.000	1,11
3	3,0	10.000	1,00
4	3,2	10.000	0,94
5	3,3	10.000	0,91
6	3,4	10.000	0,88
7	3,4	10.000	0,88
8	3,4	10.000	0,88
9	3,2	10.000	0,94
10	3,5	10.000	0,86
11	3,5	10.000	0,86
12	3,5	10.000	0,86
13	3,6	10.000	0,83
14	3,6	10.000	0,83
15	3,6	10.000	0,83
16	3,6	10.000	0,83

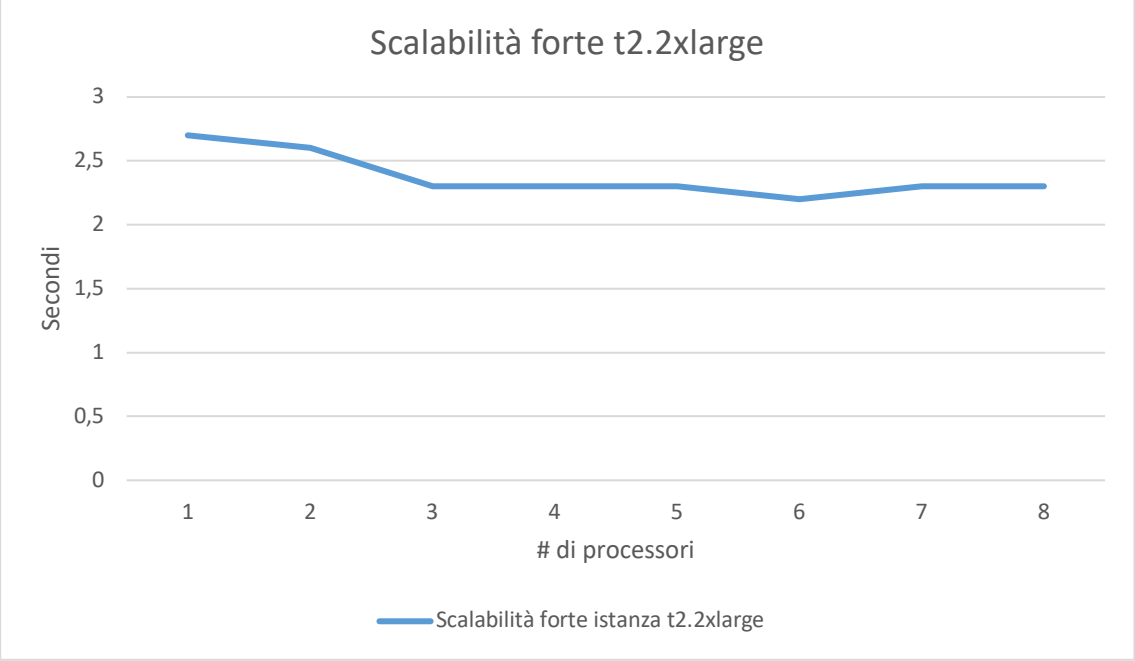


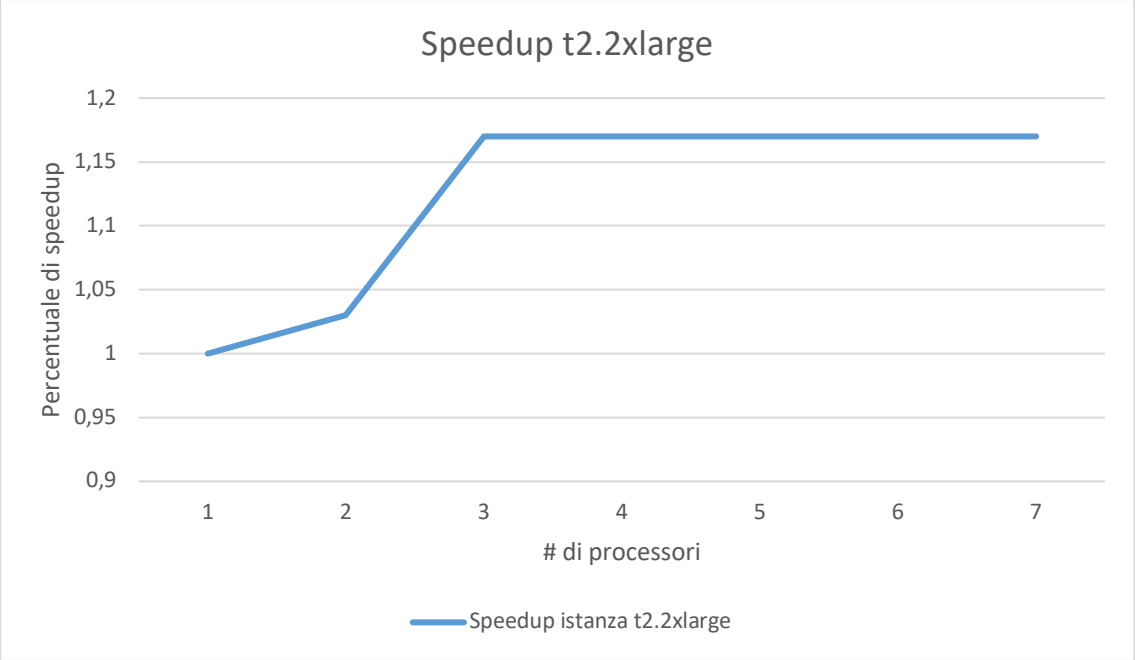


Possiamo notare che lo speedup non solo non aumenta, bensì diminuisce con l'incremento dei processori coinvolti. Si è deciso quindi di sostituire la struttura dati array con una matrice e confrontare i risultati finali.

Scalabilità forte con singola MPI_Gaterv alla fine di tutte le simulazioni ed istanza t2.2xlarge

Iterazioni=25			
vCPUs	**Tempo**	**# righe/colonne**	**Speedup**
1	2,7	10.000	1,0
2	2,6	10.000	1,03
3	2,3	10.000	1,17
4	2,3	10.000	1,17
5	2,3	10.000	1,17
6	2,2	10.000	1,17
7	2,3	10.000	1,17
8	2,3	10.000	1,17





A riprova del collo di bottiglia imposto dalla comunicazione, è stata istanziata una macchina t2.2xlarge con un totale di 8 vCPUs. Possiamo notare che, essendo i tempi di comunicazione prossimi allo 0, lo speed up aumenta fino ad assestarsi ad 1,17%. Questo vuol dire che, oltre i 3 processori, non si ha alcun ulteriore miglioramento. È possibile altresì apprezzare, anche in questo caso, come la differenza nel porre la funzione MPI_Gatherv in posizioni differenti dell'algoritmo aumenti o diminuisca i tempi per il completamento del task.

CAPITOLO 4

Conclusioni

Istruzioni per una corretta esecuzione

1. Scaricare il progetto dalla repository di GitHub:
2. Compilare il file gol.c
    a. mpicc gol.c -o gol
3. eseguire il file compilato gol usando il seguente comando
    a. mpirun -np “#processori” “hostfile” gol 100 500 3 25

       dove:

       100 è il numero di righe

       500 è il numero di colonne

       3 è il seme per la generazione della matrice random

       25 sono il totale delle iterazioni da eseguire