# Progetto PCPC Musto Emanuele Ugo

## Come far partire il codice

per compilare il codice:

`mpicc gol.c -o gol`

per runnare:

`mpirun -np N gol`

dove **N** è il numero di processori che si vuole usare

## Presentazione del codice

Il codice riportato nella repository è una soluzione parallela al problema Games of Life utilizzando la libreria MPI.

Qui sotto riporto le prestazioni che l'applicativo ha riportato più qualche considerazione. Il programma è suddiviso in varie funzioni:

```
// Funzione per inizializzare una linea di celle vive nella griglia del mondo
void seed_line(char *world, int row, int cols, int start_row, int start_col) {
    // Inizializza tutta la griglia con celle morte
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < cols; j++) {
            world[i * cols + j] = 'd';
        }
    }

    // Imposta le celle nella riga specificata come vive
    for (int i = start_col; i < start_col + 3; i++) {
        world[i + start_row * cols] = 'a';
    }
}
```

La funzione `seed_line` viene utilizzata per inizializzare una riga di celle vive ('a' alive) e quelle morte ('d' dead) nella griglia. Innanzitutto, viene inizializzata tutta la griglia con celle morte. Quindi, vengono impostate come vive le celle nella riga specificata.

```
// Funzione per inizializzare un glider nella griglia del mondo
void seed_glider(char *world, int row, int cols, int start_row, int start_col) {
    // Inizializza tutta la griglia con celle morte
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < cols; j++) {
            world[i * cols + j] = 'd';
        }
    }
```

```
    // Imposta le celle del glider come vive
    world[start_col + start_row * cols] = 'a';
    world[1 + start_col + (start_row + 1) * cols] = 'a';

    // Chiama la funzione seed_line per completare il glider
    seed_line(world, row, cols, start_row + 2, start_col - 1);
}
```

La funzione `seed_glider` viene utilizzata per inizializzare un glider nella griglia del mondo. Anche qui, viene inizializzata tutta la griglia con celle morte. Successivamente, vengono impostate come vive le celle del glider. Infine, viene chiamata la funzione `seed_line` per completare la configurazione del glider.

```
// Funzione per stampare lo stato corrente della griglia del mondo
void print_world(char *world, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (world[j + i * cols] == 'd') {
                printf("|   "); // Cella morta
            } else {
                printf("| - "); // Cella viva
            }
            if (j == cols - 1) {
                printf("|\n");
            }
        }
    }
    printf("\n\n");
}
```

la funzione `seed_glider` sempre semplicemente a stampare la griglia tra una iterazione e l'altra, se la griglia messa nella funzione main non è troppo grande.

```
// Funzione per calcolare il numero di vicini vivi di una cella nella griglia del
mondo
int count_neighbors(char *world, int row_idx, int col_idx, int rows, int cols) {
    int count = 0;

    for (int i = row_idx - 1; i <= row_idx + 1; i++) {
        for (int j = col_idx - 1; j <= col_idx + 1; j++) {
            if (i < 0 || j < 0 || i >= rows || j >= cols) {
                continue;
            } else if (i == row_idx && j == col_idx) {
                continue;
            } else if (world[j + i * cols] == 'a') {
                count++;
            }
        }
    }
```

```
    return count;
}
```

La funzione `count_neighbors` viene utilizzata per calcolare il numero di vicini vivi di una cella nella griglia del mondo. Itera sulle celle adiacenti alla cella di destinazione e conta il numero di celle vive.

```
// Funzione per calcolare lo stato successivo del mondo per un dato round
void compute_next_round(int me, int nproc, char *world, int rows, int cols) {
    // Alloca una griglia temporanea per memorizzare lo stato successivo del mondo
    char *world_tmp = malloc(rows * cols * sizeof(char));

    int local_count;
    int a, b;
    // Calcola gli indici di inizio e fine per la porzione locale della griglia as
segnata a ciascun processo
    if (me == 0) {
        a = 0;
        b = rows - 1;
    } else if (me == nproc - 1) {
        a = 1;
        b = rows;
    } else {
        a = 1;
        b = rows - 1;
    }

    // Calcola il nuovo stato per ciascuna cella nella porzione locale della grigl
ia
    for (int i = a; i < b; i++) {
        for (int j = 0; j < cols; j++) {
            // Calcola ilnumero di vicini vivi per la cella corrente utilizzando l
a funzione `count_neighbors`
            int count = count_neighbors(world, i, j, rows, cols);

            // Applica le regole del Gioco della Vita per determinare lo stato suc
cessivo della cella
            if (world[j + i * cols] == 'a') {
                if (count < 2 || count > 3) {
                    world_tmp[j + i * cols] = 'd'; // Cella muore
                } else {
                    world_tmp[j + i * cols] = 'a'; // Cella sopravvive
                }
            } else {
                if (count == 3) {
                    world_tmp[j + i * cols] = 'a'; // Cella viene creata
                } else {
                    world_tmp[j + i * cols] = 'd'; // Cella rimane morta
                }
            }
        }
```

```
    }

    // Raccoglie i risultati da tutti i processi
    char *recv_buf = NULL;
    int recv_count = rows * cols / nproc;
    if (me == 0) {
        recv_buf = malloc(rows * cols * sizeof(char));
    }
    MPI_Gather(world_tmp + cols, recv_count, MPI_CHAR, recv_buf, recv_count, MPI_C
HAR, 0, MPI_COMM_WORLD);

    // Aggiorna il mondo con i dati ricevuti
    if (me == 0) {
        for (int i = 0; i < rows * cols; i++) {
            world[i] = recv_buf[i];
        }
        free(recv_buf);
    }

    free(world_tmp);
}
```

La funzione `compute_next_round` viene utilizzata per calcolare lo stato successivo del mondo per un dato round. Prima di tutto, viene allocata una griglia temporanea per memorizzare lo stato successivo. Successivamente, vengono calcolati gli indici di inizio e fine per la porzione locale della griglia assegnata a ciascun processo. La funzione itera sulla porzione locale della griglia e calcola il nuovo stato per ogni cella utilizzando le regole del Gioco della Vita. Infine, i risultati vengono raccolti da tutti i processi utilizzando `MPI_Gather` e il mondo viene aggiornato con i dati ricevuti.

```
int main(int argc, char **argv) {
    // Inizializza MPI
    MPI_Init(&argc, &argv);
    int me, nproc;
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    // Definisce il numero di righe e colonne della griglia del mondo
    int rows = 1024;
    int cols = 1024;

    // Calcola il numero di righe assegnate a ciascun processo
    int local_rows = rows / nproc;

    // Alloca la griglia del mondo e la griglia locale
    char *world = malloc(rows * cols * sizeof(char));
    char *local_world = malloc(local_rows * cols * sizeof(char));

    // Inizializza il mondo
    if (me == 0) {
```

```
        seed_glider(world, rows, cols, 2, 2);
    }

    // Distribuisce il mondo iniziale a tutti i processi
    MPI_Scatter(world + cols, local_rows * cols, MPI_CHAR, local_world, local_rows
* cols, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Calcola e aggiorna lo stato del mondo per ogni round
    for (int round = 0; round < 25; round++) {
        compute_next_round(me, nproc, local_world, local_rows, cols);

        // Sincronizza tutti i processi
        MPI_Barrier(MPI_COMM_WORLD);

        // Raccoglie e aggiorna il mondo completo
        MPI_Gather(local_world, local_rows * cols, MPI_CHAR, world + cols, local_r
ows * cols, MPI_CHAR, 0, MPI_COMM_WORLD);

        // Stampa il mondo alla fine di ogni round
        if (rows >= 50 && cols >= 50 && me == 0) {
            printf("Round %d:\n", round + 1);
        }

        // Distribuisce il mondo aggiornato a tutti i processi
        MPI_Scatter(world + cols, local_rows * cols, MPI_CHAR, local_world, local_
rows * cols, MPI_CHAR, 0, MPI_COMM_WORLD);
    }

    // Libera la memoria
    free(world);
    free(local_world);

    // Finalizza MPI
    MPI_Finalize();

    return 0;
}
```

Nella funzione `main`, il programma inizia con l'inizializzazione di MPI e l'ottenimento del rango del processo corrente (`me`) e del numero totale di processi (`nproc`). Viene quindi definito il numero di righe e colonne della griglia del mondo, e viene calcolato il numero di righe assegnate a ciascun processo. Successivamente, vengono allocate le griglie del mondo e la griglia locale.

Il mondo viene inizializzato nel processo 0 chiamando la funzione `seed_glider` per impostare un pattern a forma di glider nella griglia.

La griglia del mondo iniziale viene distribuita a tutti i processi utilizzando `MPI_Scatter`. Il ciclo principale del programma calcola e aggiorna lo stato del mondo per ogni round, utilizzando la funzione `compute_next_round`. Vengono utilizzate operazioni di sincronizzazione come `MPI_Barrier` per garantire che tutti i processi siano allineati prima di procedere.

Alla fine di ogni round, il mondo completo viene raccolto nel processo 0 utilizzando `MPI_Gather`. Il mondo aggiornato viene quindi distribuito a tutti i processi utilizzando `MPI_Scatter` per prepararsi per il round successivo.

Infine, la memoria allocata viene liberata e MPI viene finalizzato.

## Analisi Prestazioni

Tutti i test sono stati fatti su un cluster Google Cloud Platform e2-standard-16 con 16 vCPU

### Strong scalability

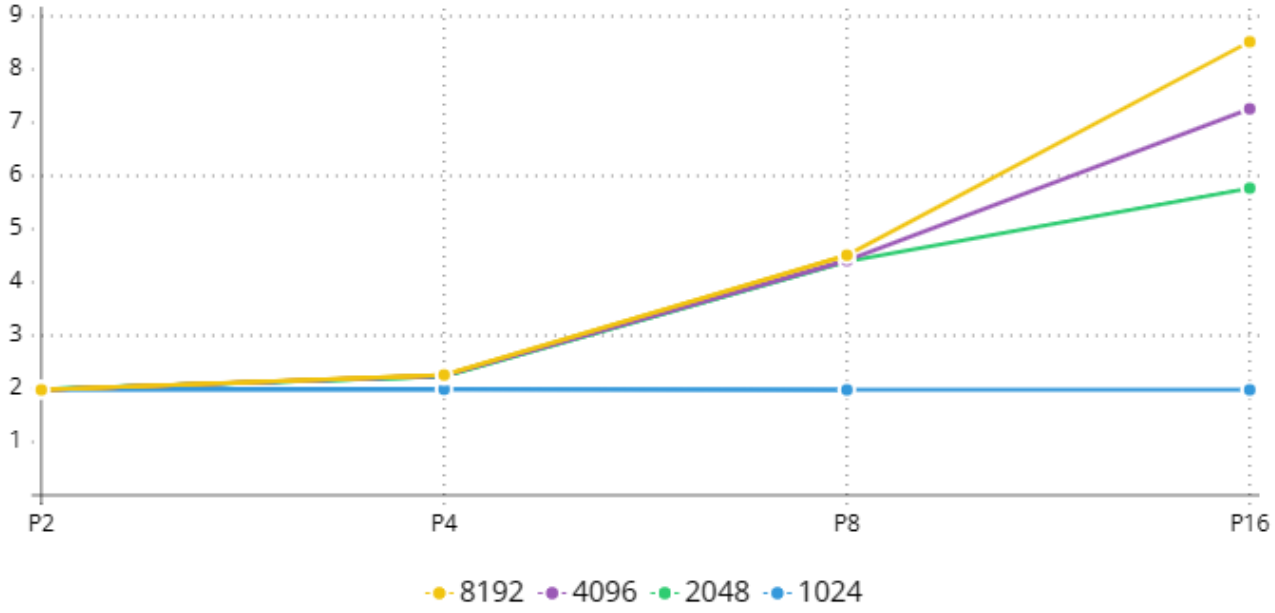
La strong scalability è dominata dalla legge di Amdhal, essa pone un limite superiore allo speedup. ### Speedup = 1/ ( s + p/n ) = T(1,size)/ T(n ,size)

Nella tabella qui sotto presente registro i dati da me registrati:

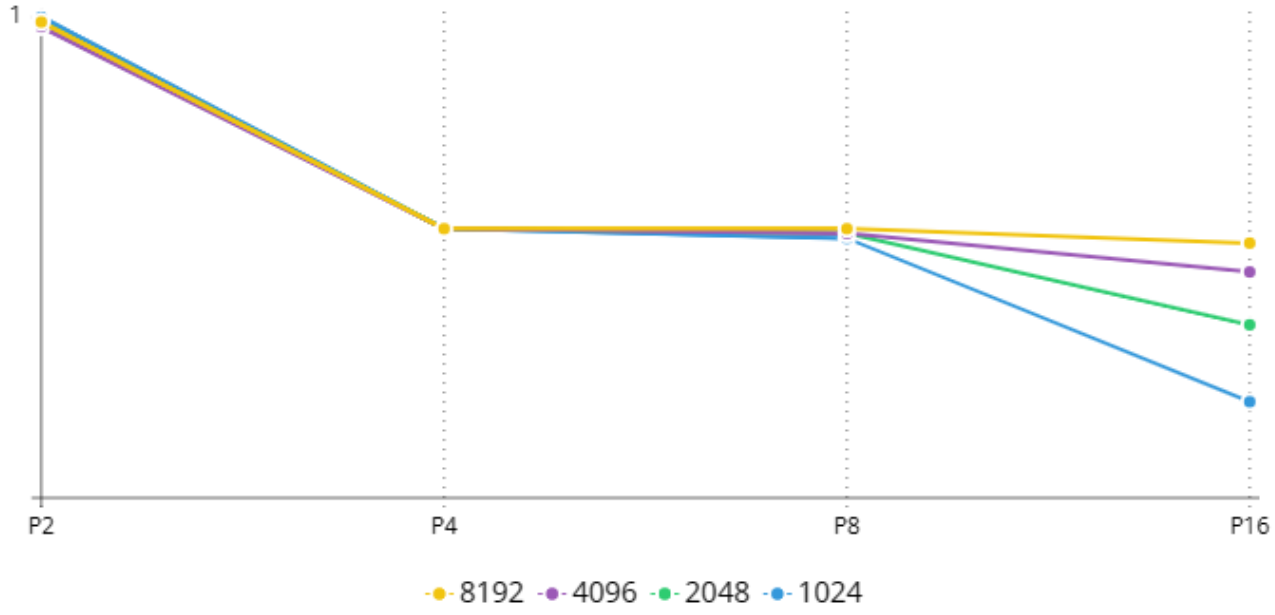
| | Sequenziale | P2 | | | P4 | | |
|---|---|---|---|---|---|---|---|
| N | tempo (s) | tempo (s) | Sp | Ep | tempo (s) | Sp | Ep |
| 1024*1024 | 4,02 | 2,01 | 1,99 | 1,00 | 1,53 | 2,23 | 0,56 |
| 2048*2048 | 14,21 | 7,12 | 1,99 | 0,99 | 6,07 | 2,23 | 0,56 |
| 4096*4096 | 58,58 | 29,41 | 1,98 | 0,98 | 24,24 | 2,25 | 0,56 |
| 8192*8192 | 251,68 | 126,01 | 1,99 | 0,99 | 97,96 | 2,26 | 0,56 |

| P8 | | | P16 | | |
|---|---|---|---|---|---|
| tempo (s) | Sp | Ep | tempo (s) | Sp | Ep |
| 0,79 | 4,32 | 0,54 | 1,07 | 3,19 | 0,20 |
| 3,11 | 4,39 | 0,55 | 2,37 | 5,76 | 0,36 |
| 12,37 | 4,41 | 0,55 | 7,26 | 7,52 | 0,47 |
| 49,04 | 4,51 | 0,56 | 25,96 | 8,52 | 0,53 |

## Speedup forte




## Efficienza



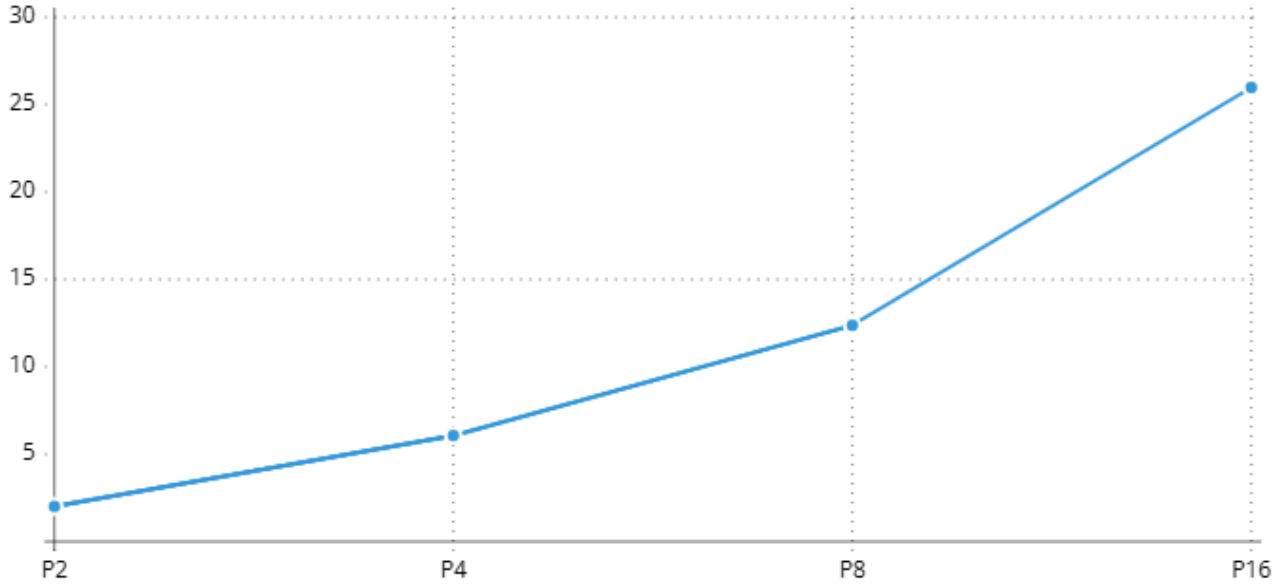
### Weak Scalability

La weak scalability è dominata dalla legge di Gustafson, essa mette in relazione la dimensione del problema con il numero di processori, infatti lo speedup ottenuto è detto anche scaled-speedup.

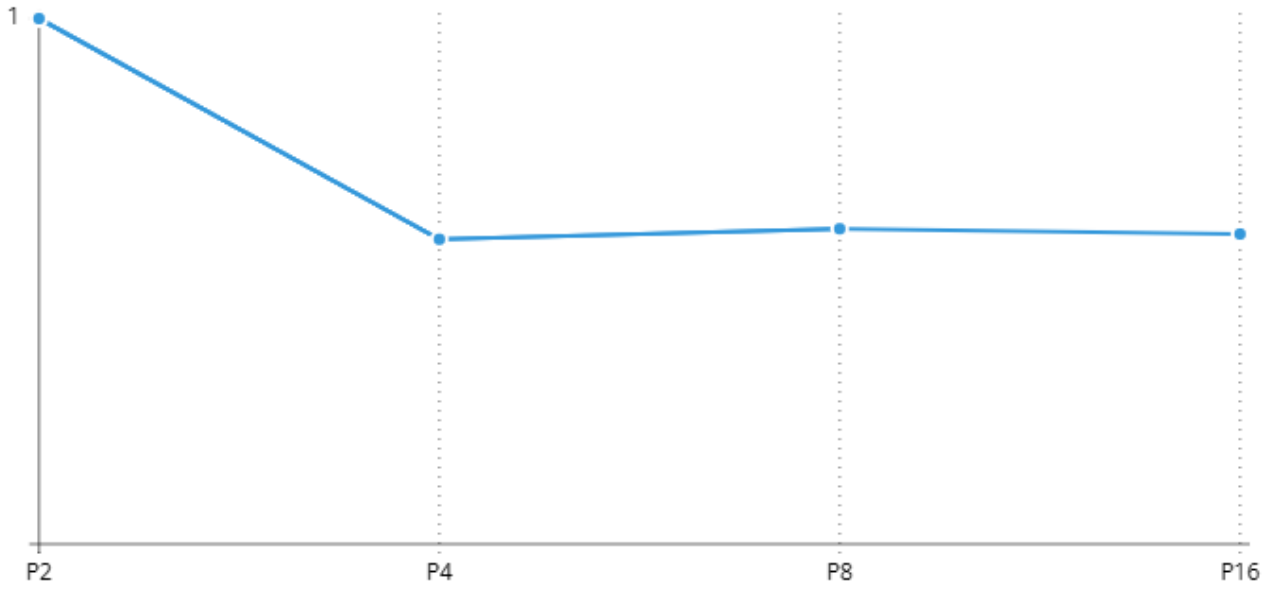
**Speedup scalato = s + p x N = N( T(1,size) ) / T( N , N * size )**

| | 1024*1024 | 2048*2048 | 4096*4096 | 8192*8192 |
|---|---|---|---|---|
| nproc | 2 | 4 | 8 | 16 |
| tempo | 2,01 | 6,07 | 12,37 | 25,96 |
| speedup | 2,01 | 2,34 | 4,73 | 9,68 |
| efficienza | 1 | 0,58 | 0,60 | 0,60 |

## Speedup debole




## Efficienza



## Conclusioni

In generale, i risultati indicano che l'aumento del numero di processi non porta a un miglioramento lineare delle prestazioni a causa dell'overhead di comunicazione. L'efficienza diminuisce man mano che si aumenta il numero di processi, la programmazione parallela per questo problema è efficiente all'aumentare della grandezza della tabella in cui si opera, più sono grandi N e M maggiore è lo speedup.