




# Game Of Life - MPI

---

Implementazione del modello matematico *Game of life* (John Conway - 1970) con linguaggio C e OpenMPI.

---

 Università degli Studi di Salerno  
 Dipartimento di Informatica  
 Programmazione Parallela e Concorrente su Cloud 2022/2023

---

 Francesco Pio Covino

## Soluzione proposta

Game of life è un modello matematico creato da John Conway nel 1970. La seguente è un'implementazione in C con l'utilizzo di OpenMPI. La soluzione proposta si può sintetizzare nei seguenti punti:

- La matrice viene allocata in porzioni di memoria contigue e quindi fisicamente è un array ma viene trattata logicamente come una matrice.
- La comunicazione fra processi segue il pattern ad anello o toroidale, il successore del processo con rank  $n-1$  avrà rank 0 e di conseguenza il predecessore del processo con rank 0 avrà rank  $n-1$ .
- La matrice  $N \times M$  viene allocata dal processo MASTER ed inizializzata da tutti i  $P$  processi in parallelo. Ogni processo inizializza un insieme di righe, di default sono  $N/\text{numero\_di\_processi}$  ma in caso di divisione non equa le righe in più vengono ridistribuite. Le righe inizializzate vengono poi re-inviate al processo MASTER attraverso l'uso della routine `MPI_Gatherv`.
- La matrice inizializzata viene suddivisa in modo equo tra tutti i processi, MASTER compreso, attraverso la routine `MPI_Scatterv`.
- I processi, in modalità asincrona, eseguono le seguenti operazioni:
  - calcolano il rank del proprio successore e predecessore nell'anello.
  - inviano (`MPI_Isend`) la loro prima riga al loro predecessore e la loro ultima riga al loro successore.
  - Si preparano a ricevere (`MPI_Irecv`) la riga di bordo superiore dal loro predecessore e la riga di bordo inferiore dal loro successore.
  - Mentre attendono le righe di bordo, iniziano ad eseguire l'algoritmo di GameofLife nelle zone della matrice in cui gli altri processi non sono coinvolti (essenzialmente tutte le righe escluse la prima e l'ultima).
  - Ricevute le righe di bordo, i processi eseguono l'algoritmo anche sulle celle che necessitano dell'utilizzo di quest'ultime.
- Le righe risultanti vengono re-inviate al MASTER attraverso l'uso della routine `MPI_Gatherv`.

## Dettagli Implementativi

Di seguito alcuni degli aspetti più importanti dell'implementazione.

## Analisi degli argomenti

In base al numero di argomenti inseriti dall'utente, viene scelto la variante da eseguire: nel primo caso il processo MASTER riempie la matrice iniziale con un file pattern, nel secondo caso le dimensioni sono scelte dall'utente e nel terzo caso vengono utilizzate quelle di default.

```
switch (argc)
{
    case 3: /* l'utente ha indicato un pattern da file */
        is_file = true;
        if (rank == MASTER) {
            /* preparazione file */
            dir = "patterns/";
            filename = argv[1];
            ext = ".txt";
            file = malloc(strlen(dir) + strlen(filename) + strlen(ext) + 1);
            sprintf(file, "%s%s%s", dir, filename, ext);
            printf("--Generate matrix seed from %s--\n", file);
            /* il processo master imposta le dimensioni della matrice in base al
file di input */
            check_matrix_size(file, &row_size, &col_size);
        }
        /* master invia la size della matrice a tutti i processi */
        MPI_Bcast(&row_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
        MPI_Bcast(&col_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
        iterations = atoi(argv[2]);
        break;
    case 4: /* le dimensioni sono scelte dall'utente */
        row_size = atoi(argv[1]);
        col_size = atoi(argv[2]);
        iterations = atoi(argv[3]);
        break;
    case 1: /* esegue l'algoritmo con le configurazioni di default */
        row_size = DEF_ROWS;
        col_size = DEF_COLS;
        iterations = DEF_ITERATION;
        break;
    default:
        printf("Error, check the number of arguments.\n");
        MPI_Finalize();
        return 0;
        break;
}
```

## Nuovo tipo di dato MPI

Viene creato un nuovo tipo di dato MPI replicando `MPI_CHAR` `col_size` volte in posizioni contigue (`col_size` indica il numero di colonne della matrice). La contiguità permette di migliorare le prestazioni nel passaggio di dati di grandi dimensioni.

```
MPI_Type_contiguous(col_size, MPI_CHAR, &mat_row);
MPI_Type_commit(&mat_row);
```

## Divisione del carico

La matrice viene equamente divisa, per numero di righe, tra i processi. Nel caso di divisione non equa, i primi resto processi ricevono una riga in più.

```
/* ogni cella k memorizza il numero di righe assegnate al processo k */
rows_for_proc = calloc(num_proc, sizeof(int));
/* ogni cella k memorizza il displacement da applicare al processo k */
displacement = calloc(num_proc, sizeof(int));
/* divisione delle righe */
int base = (int)row_size / num_proc;
int rest = row_size % num_proc;
/* righe già assegnate */
int assigned = 0;

/* calcolo righe e displacement per ogni processo */
for (int i = 0; i < num_proc; i++) {
    displacement[i] = assigned;
    /* nel caso di resto presente, i primi resto processi ricevono una riga in più*/
    if (rest > 0) {
        rows_for_proc[i] = base + 1;
        rest--;
    } else {
        rows_for_proc[i] = base;
    }
    assigned += rows_for_proc[i];
}
```

Ogni processo alloca la sua porzione di righe, il cui numero è stato calcolato nella fase precedente:

```
result_buff = calloc(rows_for_proc[rank] * col_size, sizeof(char));
```

Ogni processo inizializza le celle delle sue righe (ALIVE o DEAD) in modo casuale e le invia al processo MASTER. La routine `MPI_Gatherv` raccoglie i dati da tutti i processi e li concatena nel buffer del processo MASTER.

```
MPI_Gatherv(result_buff, rows_for_proc[rank], mat_row, matrix, rows_for_proc,
displacement, mat_row, MASTER, MPI_COMM_WORLD);
```

Ogni processo alloca i propri buffer:

1. un buffer per la ricezione della sottomatrice
2. un buffer per memorizzare la nuova matrice a seguito del calcolo della nuova generazione
3. un buffer per ricevere la riga precedente alle proprie
4. un buffer per ricevere la riga successiva alle proprie

```
recv_buff = calloc(rows_for_proc[rank] * col_size, sizeof(char));
result_buff = calloc(rows_for_proc[rank] * col_size, sizeof(char));
prev_row = calloc(col_size, sizeof(char));
next_row = calloc(col_size, sizeof(char));
```

## Calcolo delle generazioni

Le seguenti porzioni di codice vengono eseguite per un numero  $I$  di iterazioni scelte dall'utente:

La matrice, precedentemente inizializzata, viene distribuita a tutti i processi con la routine `MPI_ScatterV`.

```
MPI_Scatterv(matrix, rows_for_proc, displacement, mat_row, recv_buff,
rows_for_proc[rank], mat_row, MASTER, MPI_COMM_WORLD);
```

Ogni processo, in modalità asincrona, attraverso le routine `MPI_Isend` e `MPI_Irecv`:

1. invia al suo predecessore la sua prima riga
2. si mette in attesa della riga precedente alle sue
3. invia al suo successore la sua ultima riga
4. attende la riga successiva alle sue

```
MPI_Isend(recv_buff, 1, mat_row, prev, TAG_NEXT, MPI_COMM_WORLD, &send_request);
MPI_Irecv(prev_row, 1, mat_row, prev, TAG_PREV, MPI_COMM_WORLD, &prev_request);
MPI_Isend(recv_buff + (col_size * (sub_rows_size - 1)), 1, mat_row, next,
TAG_PREV, MPI_COMM_WORLD, &send_request);
MPI_Irecv(next_row, 1, mat_row, next, TAG_NEXT, MPI_COMM_WORLD, &next_request);
```

Ogni processo calcola il valore delle celle che non necessitano dei valori posseduti da altri processi. Vengono pertanto escluse la prima riga e l'ultima. Per ogni cella target vengono calcolati i vicini ALIVE nell'intorno (le 8 celle che lo racchiudono). Il dato appena calcolato viene poi utilizzato per calcolare il nuovo valore della cella.

```
void compute(char* origin_buff, char* result_buff, int row_size, int col_size) {
    for (int i = 1; i < row_size - 1; i++) {
        for (int j = 0; j < col_size; j++) {

            /* memorizza i vicini vivi nell'intorno della cella target */
            int live_count = 0;
            for (int row = i - 1; row < i + 2; row++) {
                for (int col = j - 1; col < j + 2; col++) {
                    if (row == i && col == j) {
```

```

                continue;
            }
            if (origin_buff[row * col_size + (col % col_size)] ==
ALIVE) {
                live_count++;
            }
        }
        /* decide lo stato della cella per la generazione successiva */
        life(origin_buff, result_buff, i * col_size + j, live_count);
    }
}

```

Nello specifico la funzione `life()` controlla le 3 casistiche con un semplice If-Else: una cellula ALIVE con 2 o 3 vicini ALIVE sopravvive per la prossima generazione, se è DEAD e ha 3 vicini ALIVE diventa ALIVE nella prossima generazione. Altrimenti in tutti i restanti casi il suo stato è DEAD.

```

void life(char *origin, char *result, int index, int live_count) {
    if (origin[index] == ALIVE && (live_count == 2 || live_count == 3)) {
        result[index] = ALIVE;
    } else if (origin[index] == DEAD && live_count == 3) {
        result[index] = ALIVE;
    } else {
        result[index] = DEAD;
    }
}

```

Prima di calcolare i valori delle celle che necessitano delle righe degli altri processi, si attende il completamento e la ricezione delle righe di bordo con la routine `MPI_Waitany`.

```

MPI_Request to_wait[] = {prev_request, next_request};
int handle_index;
MPI_Waitany(2, to_wait, &handle_index, &status);

```

Una volta completate entrambe le richieste, vengono calcolati i valori delle celle mancanti utilizzando le righe di frontiera appena ricevute. La riga -1 identifica la riga precedente ricevuta dal processo precedente. Per raggiungere lo scopo basterà controllare la riga precedente e le prime due assegnate al processo corrente. Vengono calcolati i vicini ALIVE nell'intorno della cella target ed eseguito il calcolo per identificare il suo stato nella successiva generazione.

```

void compute_prev(char* origin_buff, char* result_buff, char* prev_row, int
col_size) {
    for (int j = 0; j < col_size; j++) {
        int live_count = 0;
    }
}

```

```

        for (int row = -1; row < 2; row++) {
            for (int col = j - 1; col < j + 2; col++) {
                if (row == 0 && col == j)
                    continue;
                /* controlla la riga precedente */
                if (row == -1) {
                    if (prev_row[col % col_size] == ALIVE)
                        live_count++;
                } else { /* altrimenti controlla nella sotto-matrice */
                    if (origin_buff[row * col_size + (col % col_size)] == ALIVE)
                        live_count++;
                }
            }
        }
        /* decide lo stato della cella */
        life(origin_buff, result_buff, j, live_count);
    }
}

```

Allo stesso modo vengono poi calcolati i valori delle righe superiori. La riga `row_size` identifica la riga successiva ricevuta dal processo successivo. Per raggiungere lo scopo basterà controllare le ultime due righe e quella successiva dapprima controllando i vicini ALIVE nell'intorno e poi eseguendo la funzione `life()`.

```

void compute_next(char* origin_buff, char* result_buff, char* next_row, int
row_size, int col_size) {
    for (int j = 0; j < col_size; j++) {
        int live_count = 0;
        for (int row = row_size - 2; row < row_size + 1; row++) {
            for (int col = j - 1; col < j + 2; col++) {
                /* se sto analizzando la cella corrente continuo */
                if (row == row_size - 1 && col == j)
                    continue;
                /* controlla la riga successiva o la matrice in base al caso */
                if (row == row_size) {
                    if (next_row[col % col_size] == ALIVE)
                        live_count++;
                } else {
                    if (origin_buff[row * col_size + (col % col_size)] ==
ALIVE)
                        live_count++;
                }
            }
        }
        /* decide lo stato della cella nella posizione indicata */
        life(origin_buff, result_buff, (row_size - 1) * col_size + j, live_count);
    }
}

```

Le righe, con i valori aggiornati per la prossima generazione, vengono re-inviolate al processo MASTER e concatenate nel buffer con la routine `MPI_Gatherv`:

```
MPI_Gatherv(result_buff, rows_for_proc[rank], mat_row, matrix, rows_for_proc,
displacement, mat_row, MASTER, MPI_COMM_WORLD);
```

## Compilazione ed esecuzione

Sono presenti due versioni dell'algoritmo, una versione sequenziale ed una parallela, entrambe utilizzano OpenMPI. La versione sequenziale utilizza OpenMPI solo affinché il calcolo dei tempi sia svolto nel medesimo modo.

### ► GoL versione sequenziale

Per la compilazione eseguire il seguente comando da terminale:

```
mpicc -o gol sequential_gol.c
```

Per l'esecuzione esistono due varianti:

La prima variante permette di inizializzare la matrice seed con un pattern memorizzato su file. Il file pattern va inserito in formato plain text (.txt) nella directory *patterns/*. Una cella della matrice indicata come ALIVE conterrà il simbolo *O* mentre se indicata come DEAD conterrà il simbolo *.*; I seguenti sono due esempi di pattern:

**Pulsar pattern**

**Glidergun pattern**

```

1 .....
2 .....0..0....
3 ...000...000.....0..0....
4 .....0..0....
5 .0...0.0...0.....00...00....
6 .0...0.0...0...0..0.....0.0...0.0....
7 .0...0.0...0...0..0.....00000...00000.
8 ...000...000.....00...00.....
9 .....000...000.....
10 ...000...000.....
11 .0...0.0...0.....00000...00000.
12 .0...0.0...0..000.....000...0.0...0.0....
13 .0...0.0...0...00...00.....00...00....
14 .....0..0.....0..0....
15 ...000...000.....0..0.....0..0....
16 .....0..0....
17 .....
```

```

1 .....
2 .....0.....
3 .....0.0.....
4 .....00.....00.....00....
5 .....0..0...00.....00....
6 .00.....0...0..00.....
7 .00.....0..0.00...0.0.....
8 .....0...0.....0.....
9 .....0..0.....
10 .....00.....
11 .....|
```

Oltre al nome del file va specificato il numero di iterazioni da eseguire:

```
mpirun -n 1 gol pulsar 10
```

La seconda variante permette di eseguire l'algoritmo specificando 4 argomenti:

- numero di righe
- numero di colonne
- numero di iterazioni

N.B. La matrice di partenza verrà generata in maniera casuale.

```
mpirun -n 1 gol 100 200 8
```

### ► GoL versione parallela

Per la compilazione eseguire il seguente comando da terminale:

```
mpicc -o mpigol mpi_gol.c
```

Per l'esecuzione esistono 3 varianti:

La prima variante permette di inizializzare la matrice seed con un pattern memorizzato su file. Il file pattern va inserito in formato plain text (.txt) nella directory *patterns/*. Una cella della matrice indicata come viva conterrà il simbolo *O* mentre se indicata come morta conterrà il simbolo *.*; I seguenti sono due esempi di pattern:

**Pulsar pattern**

**Glidergun pattern**

```

1 .....
2 .....0..0....
3 ..000..000.....0..0....
4 .....0..0....
5 .0...0.0...0.....00..00...
6 .0...0.0...0...0..0.....0.0..0.0...
7 .0...0.0...0...0..0.....00000...00000.
8 ..000..000.....00..00.....
9 .....000...000.....
10 ..000..000.....
11 .0...0.0...0.....00000...00000.
12 .0...0.0...0..000.....000...0.0..0.0...
13 .0...0.0...0...00..00.....00..00...
14 .....0..0.....0..0....
15 ..000..000.....0..0.....0..0....
16 .....0..0....
17 .....
```

```

1 .....
2 .....0.....
3 .....0.0.....
4 .....00.....00.....00...
5 .....0..0...00.....00...
6 .00.....0...0..00.....
7 .00.....0..0.00...0.0.....
8 .....0...0.....0.....
9 .....0..0.....
10 .....00.....
11 .....
```

Oltre al nome del file va specificato il numero di iterazioni da eseguire e il numero di processi da utilizzare:

```
mpirun -n 5 mpigol pulsar 10
```

La seconda variante permette di eseguire l'algoritmo specificando 4 argomenti:

- numero di processori
- numero di righe
- numero di colonne



- numero di iterazioni

N.B. La matrice di partenza verrà generata in maniera casuale.

```
mpirun -n 5 mpigol 100 200 8
```

La terza variante non richiede alcun argomento aggiuntivo oltre al numero di processori da utilizzare ed eseguirà l'algorithmo utilizzando i parametri di default (una matrice 240x160 su 6 iterazioni), riempiendo la matrice con valori casuali.

```
mpirun -n 5 mpigol
```

Correttezza

Per dimostrare la correttezza della soluzione sono stati utilizzati due pattern noti, *pulsar* e *glidergun*.

Pulsar

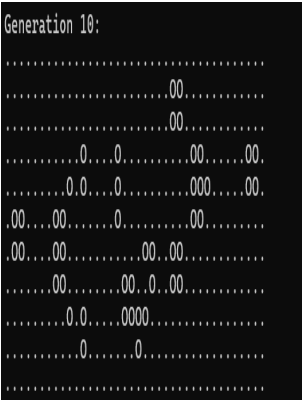
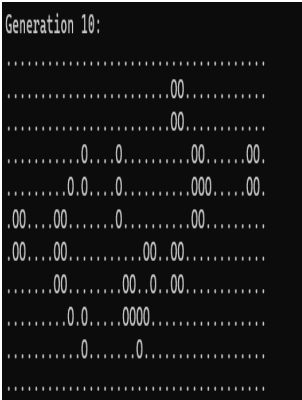
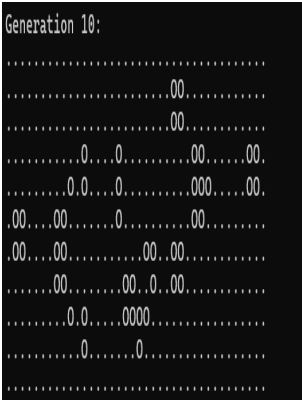
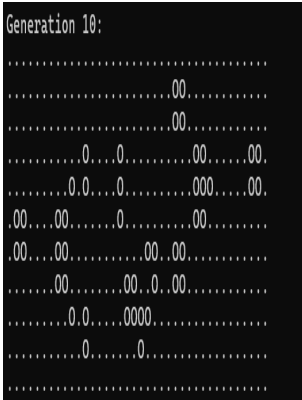


GliderGun



Il risultato è stato confrontato con una simulazione testabile [qui](#). Il programma è stato eseguito per un numero fisso di iterazioni pari a 10 e variando il numero di processi utilizzati. Di seguito i risultati:

Risultati Pattern GliderGun

ver. sequenziale	2 processi	4 processi	8 processi
			

## Risultati Pattern Pulsar

ver. sequenziale	2 processi	4 processi	8 processi
<pre> Generation 10: ..... ...0...0..... ...0...0.....000.000... ...00..00.....00.00... .....00.00..... 000..00.00..000.....0...0...0... ..0.0.0.0.0.0.....0000.....0000.....0000.. ...00..00.....0.0.....000.....000.. .....000..000..... ...00..00.....0.....0.....000.....000.. ..0.0.0.0.0.0.....0.....0.....0000.....0000.. 000..00.00..000..000..000..0...0...0...0... .....0.0.....00.00..... ...00..00.....0000.....00.00..... ...0...0.....000.000..... ...0...0..... </pre>	<pre> Generation 10: ..... ...0...0..... ...0...0.....000.000... ...00..00.....00.00... .....00.00..... 000..00.00..000.....0...0...0... ..0.0.0.0.0.0.....0000.....0000.....0000.. ...00..00.....0.0.....000.....000.. .....000..000..... ...00..00.....0.....0.....000.....000.. ..0.0.0.0.0.0.....0.....0.....0000.....0000.. 000..00.00..000..000..000..0...0...0...0... .....0.0.....00.00..... ...00..00.....0000.....00.00..... ...0...0.....000.000..... ...0...0..... </pre>	<pre> Generation 10: ..... ...0...0..... ...0...0.....000.000... ...00..00.....00.00... .....00.00..... 000..00.00..000.....0...0...0... ..0.0.0.0.0.0.....0000.....0000.....0000.. ...00..00.....0.0.....000.....000.. .....000..000..... ...00..00.....0.....0.....000.....000.. ..0.0.0.0.0.0.....0.....0.....0000.....0000.. 000..00.00..000..000..000..0...0...0...0... .....0.0.....00.00..... ...00..00.....0000.....00.00..... ...0...0.....000.000..... ...0...0..... </pre>	<pre> Generation 10: ..... ...0...0..... ...0...0.....000.000... ...00..00.....00.00... .....00.00..... 000..00.00..000.....0...0...0... ..0.0.0.0.0.0.....0000.....0000.....0000.. ...00..00.....0.0.....000.....000.. .....000..000..... ...00..00.....0.....0.....000.....000.. ..0.0.0.0.0.0.....0.....0.....0000.....0000.. 000..00.00..000..000..000..0...0...0...0... .....0.0.....00.00..... ...00..00.....0000.....00.00..... ...0...0.....000.000..... ...0...0..... </pre>

Tutti i test, dopo le 10 iterazioni prefissate, restituiscono lo stesso risultato confermando la correttezza della soluzione indipendentemente dal numero di processi utilizzati.

## Analisi performance

Le prestazioni sono state valutate su un cluster GoogleCloud di 4 nodi e2-standard-4 con 4vCPU (16 vCPU in totale) in termini di scalabilità forte e di scalabilità debole. Per i test sono state utilizzate matrici quadrate per una questione di semplicità.

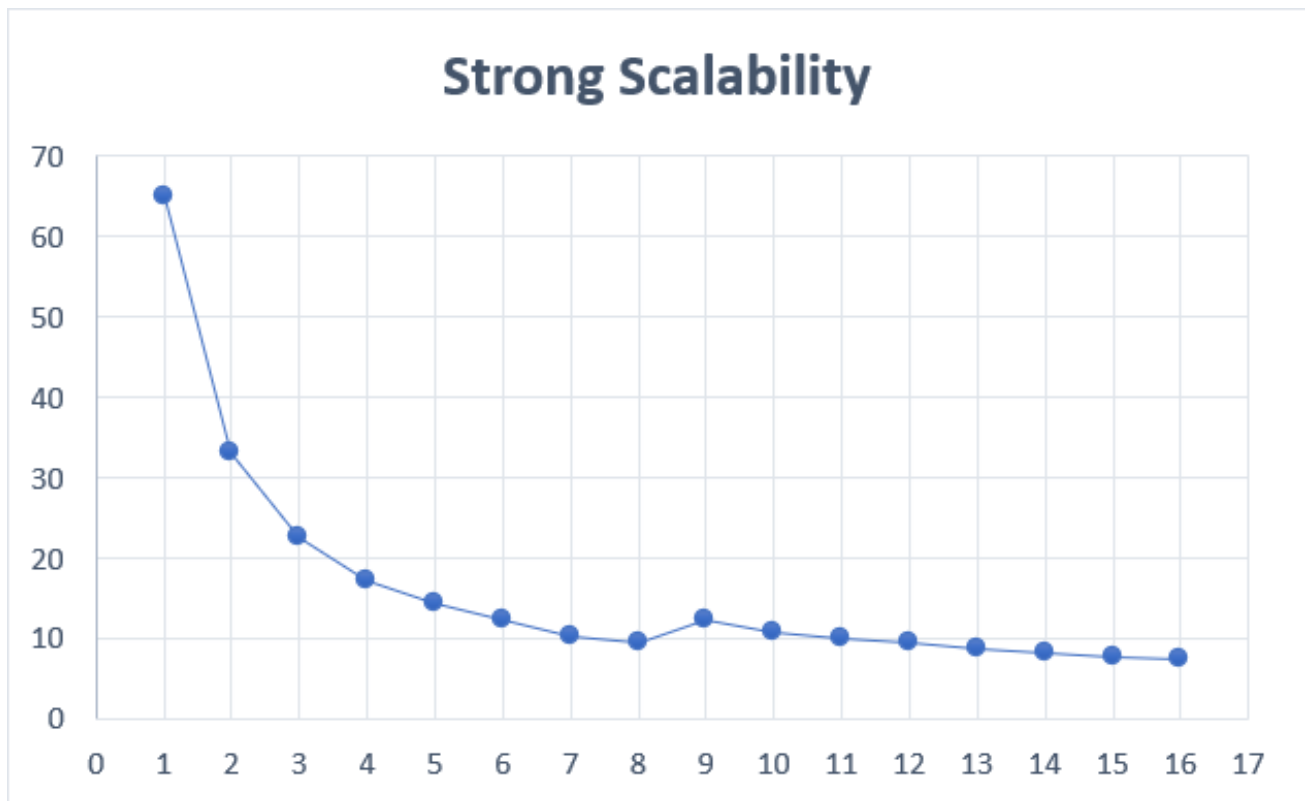
## Scalabilità forte

La scalabilità forte riguarda lo speedup per una dimensione fissa del problema rispetto al numero di processori. Per il test le dimensioni della matrice di partenza sono state fissate a 4000 righe e 4000 colonne e a variare sarà il numero di processori utilizzati (da 1 a 16). Il numero di iterazioni è stato fissato a 50. I tempi di esecuzione sono stati calcolati su una media di 3 esecuzioni. Di seguito i risultati in formato tabellare:

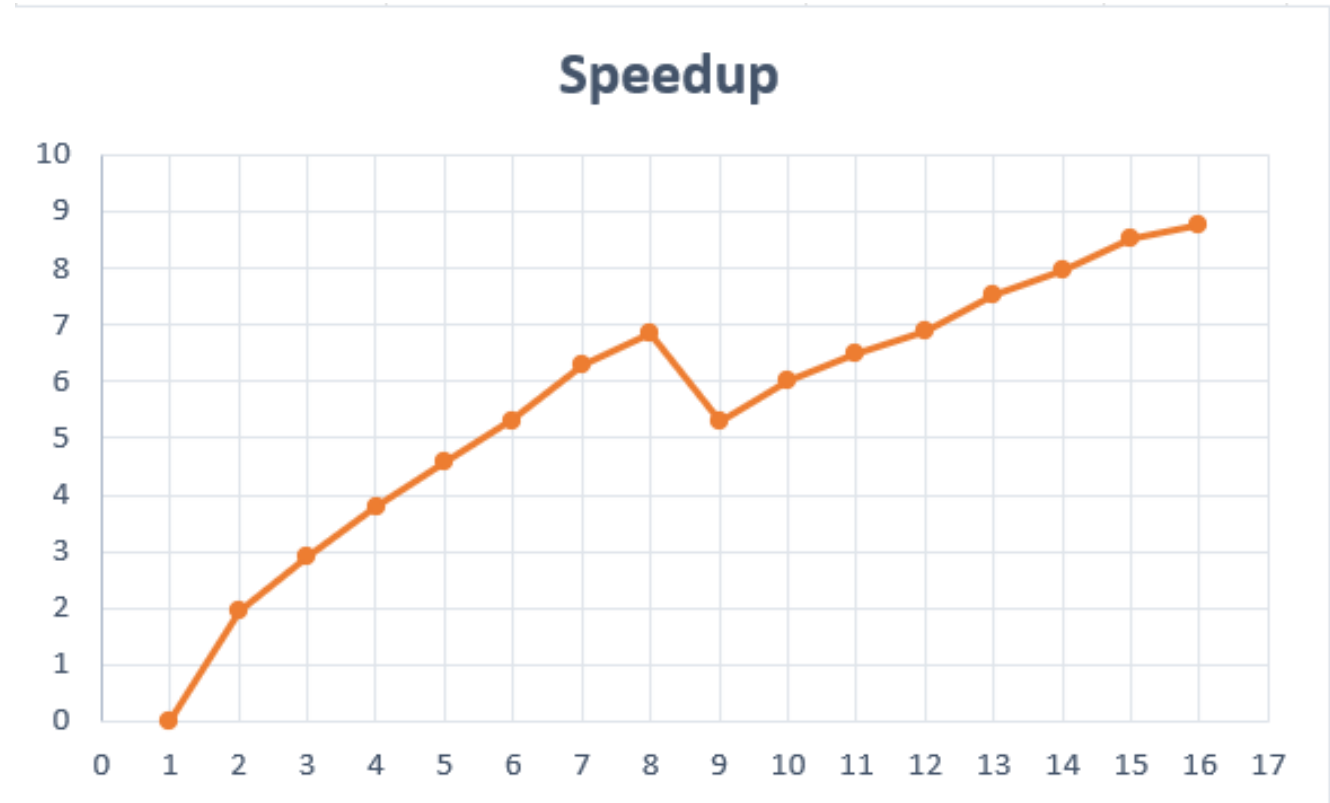
# Processori	Dim. matrice (R=C)	Tempi di esecuzione (ms)	Speedup
1	4000	64,89	//
2	4000	32,95	1,96
3	4000	22,46	2,88
4	4000	17,20	3,77
5	4000	14,20	4,56
6	4000	12,20	5,31
7	4000	10,31	6,29
8	4000	9,46	6,85
9	4000	12,27	5,28
10	4000	10,80	6,00
11	4000	10,01	6,48

# Processori	Dim. matrice (R=C)	Tempi di esecuzione (ms)	Speedup
12	4000	9,43	6,88
13	4000	8,60	7,53
14	4000	8,15	7,95
15	4000	7,59	8,54
16	4000	7,39	8,77

Di seguito invece il grafico che mostra il rapporto fra i tempi di esecuzione (ordinata) e il numero di processori utilizzati (ascissa). Inizialmente il tempo di esecuzione scende vertiginosamente fino a stabilizzarsi dagli 8 processori in poi.



Nel grafico seguente invece viene mostrato come varia lo speedup (ordinata) al crescere dei processori (ascissa) utilizzati. Inizialmente lo speedup cresce in maniera ottimale fino ad un anomalo calo con 9 processori per poi ri-stabilizzarsi con una crescita più lenta.

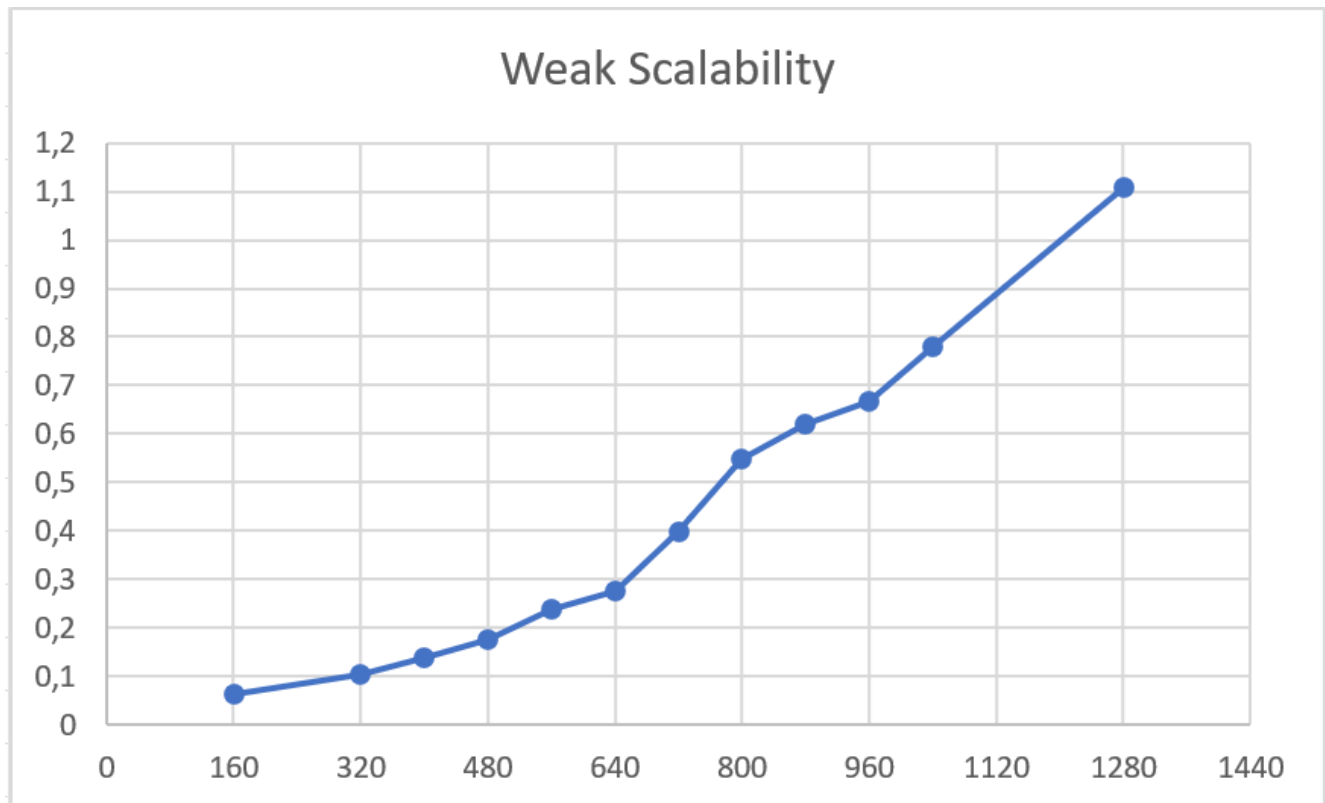


### Scalabilità debole

Lo scaling debole riguarda lo speedup per un problema di dimensioni scalari rispetto al numero di processori. Il numero di processori è stato fissato a 8 e il numero di iterazioni a 50. A variare sono le dimensioni della matrice di partenza. Dopo ogni prova aumentano il numero di righe e colonne gestite dal singolo processo (aggiunte 10 righe e 10 colonne). Di seguito i risultati sotto forma di tabella:

# Processori	Dim. matrice (R=C)	Tempi di esecuzione (ms)
8	160	0.6420
8	320	0.1048
8	400	0.1382
8	480	0.1760
8	560	0.2381
8	640	0.2771
8	720	0.3966
8	800	0.5469
8	880	0.6190
8	960	0.6670
8	1040	0.7797
8	1280	1.1082

Il grafico invece mostra il rapporto tra il tempo di esecuzione (ordinata) e la dimensione della matrice quadrata (ascissa):



## Conclusioni

Come mostrato nei risultati, la soluzione proposta scala discretamente bene. I processi lavorano parallelamente dalla prima fase dell'algoritmo, il riempimento della matrice, fino alla fase di calcolo della generazione successiva. Il processo MASTER si occupa di dividere la matrice e di riunire i risultati di ogni processo ma partecipa anch'esso alla computazione.

Per quanto riguarda la scalabilità forte, i risultati ottenuti sono inizialmente ottimi, i tempi diminuiscono, quasi dimezzano e lo speedup tende a quello ideale. Dall'utilizzo di otto processori in poi i tempi si stabilizzano e lo speedup migliora meno velocemente. Una stabilizzazione che si può imputare anche alla dimensione dell'input, non abbastanza grande per 8 processi, o all'overhead di comunicazione. Sulla base di alcuni test effettuati in locale con la medesima taglia del problema, è possibile affermare che l'overhead di comunicazione causato dall'utilizzo di un cluster remoto è impattante sui tempi di computazione, è quindi fondamentale tenere in considerazione questo particolare nella valutazione delle performance.

In termini di scalabilità debole, il programma ha ottenuto le performance attese. Per natura dell'algoritmo la computazione sfrutta molto le risorse in termini di CPU, vengono infatti analizzate indipendentemente tutte le celle della matrice e per ogni cella l'intorno. L'aumento della taglia della matrice fa diminuire di conseguenza le performance.