

-INTRODUZIONE-

L'universo di Game of Life è una griglia ortogonale bidimensionale infinita di celle quadrate, ognuna delle quali si trova in uno dei due stati possibili, vivo (A) o morto (D).

Ogni cella interagisce con i suoi otto vicini, che sono le celle adiacenti orizzontalmente, verticalmente o diagonalmente. A ogni passo nel tempo, si verificano le seguenti transizioni:

- Ogni cellula viva con meno di 2 vicini vivi muore come se fosse causata dalla sottopopolazione.
- Ogni cellula viva con 2 o 3 vicini vivi passa alla generazione successiva.
- Ogni cellula viva con più di 3 vicini vivi muore, come per sovrappopolazione.
- Ogni cellula morta con esattamente 3 vicini vivi diventa una cellula viva, come se si riproducesse.

Questo progetto si basa sulla realizzazione del modello matematico Game of Life, utilizzando e sfruttando le potenzialità di MPI. La soluzione è stata pensata per realizzare una divisione della matrice a righe. Dove ciascun processo riceverà un numero più equo possibile di righe, esegue le operazioni di aggiornamento e invia al MASTER la matrice relativa.

-SOLUZIONE-

Per la realizzazione del progetto vengono effettuate le seguenti operazioni:

- Dopo aver recuperato i parametri per la realizzazione della matrice e il numero di iterazioni, il processo MASTER (rank 0) crea e popola la matrice $N \times M$ di gioco iniziale.
- Vengono calcolati le divisioni ottimali della matrice per rendere la suddivisione del carico di lavoro equa tra i processi. Successivamente la matrice di gioco viene ripartita tra i processi, compreso il MASTER, tramite una `Scatterv`
- In un ciclo che gestisce le iterazioni del gioco, i processi comunicano in maniera asincrona per inviare la prima riga della matrice al processo precedente e l'ultima riga al processo successivo
- Una volta ottenute le righe il programma controlla se ci sono celle che possono essere calcolate senza la prima ed ultima riga. Una volta calcolate le righe intermedie passa a gestire la riga superiore ed inferiore.
- Terminate le iterazioni, i processi inviano al nodo MASTER le relative sotto-matrici aggiornate tramite una `Gatherv`

-ESECUZIONE-

compilazione:

```
mpicc -o gol gameOfLife.c
```

esecuzione:

```
mpirun -np 5 --allow-run-as-root gol --n=100 --m=100 --i=100
```

n = numero di righe m = numero di colonne i = numero di iterazioni

-IMPLEMENTAZIONE-

FUNZIONI

checkStatus

Controlla lo stato di un elemento

Parametri

Nome	Descrizione
Element	La cella che si vuole controllare

```
int checkStatus(char element){  
    if(element == 'A'){  
        return 1;  
    }else{  
        return 0;  
    }  
}
```

exec_middle_row

Questa funzione attraversa le righe intermedie della matrice, escludendo i bordi, e calcola il nuovo stato di ogni cella basato sullo stato delle celle circostanti. Per ogni cella, vengono considerate le celle adiacenti in tutte le direzioni (orizzontale, verticale e diagonale), e il numero di celle vive adiacenti viene conteggiato utilizzando la funzione checkStatus. Quindi, l'aggiornamento della matrice next_era avviene applicando le regole del gioco.

All'interno del ciclo principale, la variabile `n_alive` tiene traccia del numero di celle vive adiacenti a una particolare cella. Le variabili `prev_col` e `next_col` vengono calcolate per gestire le condizioni di bordo e ottenere gli indici delle colonne precedente e successiva rispetto a una colonna data.

Parametri

Nome	Descrizione
<code>matrix</code>	Matrice dell'era attuale
<code>next_era</code>	Matrice della prossima era
<code>n_row</code>	Numero di righe
<code>n_cols</code>	Numero di colonne
<code>rank</code>	Rank del processo

```
void exec_middle_row(char * matrix, char *next_era, int n_row, int n_cols, int rank){
    //printf("sono %d ecco la matrice mid %d x %d: \n", rank, n_row, n_cols);
    int n_alive = 0;
    char status;
    for(int i=1; i<n_row-1; i++){
        for (int j = 0; j < n_cols; j++){
            n_alive=0;
            int prev_col = (j == 0) ? n_cols - 1 : j - 1;
            int next_col = (j == n_cols-1) ? 0 : j+1;
            status=matrix[i*n_cols+j];

            n_alive += checkStatus(matrix[i*n_cols+prev_col]);
            n_alive += checkStatus(matrix[i*n_cols+j]);
            n_alive += checkStatus(matrix[i*n_cols+next_col]);

            n_alive += checkStatus(matrix[(i+1)*n_cols+prev_col]);
            n_alive += checkStatus(matrix[(i+1)*n_cols+j]);
            n_alive += checkStatus(matrix[(i+1)*n_cols+next_col]);

            n_alive += checkStatus(matrix[(i-1)*n_cols+prev_col]);
            n_alive += checkStatus(matrix[(i-1)*n_cols+j]);
            n_alive += checkStatus(matrix[(i-1)*n_cols+next_col]);
            update_matrix(next_era, n_alive, n_cols, i, j, status);
        }
    }
}
```

update_matrix

Questa funzione implementa le regole del gioco della vita di Conway per determinare lo stato della cella nella prossima era, basandosi sullo stato attuale della cella (`status`) e sul numero di celle vive

adiacenti (n_alive).

Parametri

Nome	Descrizione
next_era_matrix	Matrice della prossima era
n_alive	Numero di celle vive adiacenti alla cella considerata
M	Numero di colonne nella matrice
i	Indice della riga della cella considerata
j	Indice della colonna della cella considerata
status	Stato attuale della cella (vivo o morto)

```
void update_matrix(char *next_era_matrix,int n_alive,int M, int i,int j,char status){
    if(status=='D'){
        if(n_alive==3){
            next_era_matrix[(i*M)+j] = 'A';
        }else{
            next_era_matrix[(i*M)+j] = 'D';
        }
    }else{
        if(n_alive<2){
            next_era_matrix[(i*M)+j] = 'D';
        }else if(n_alive ==2 || n_alive==3){
            next_era_matrix[(i*M)+j] = 'A';
        }else if(n_alive > 3){
            next_era_matrix[(i*M)+j] = 'D';
        }
    }
    //printf("%c\n",next_era_matrix[i*M+j]);
}
```

exec_last_first_rows

Questa funzione si occupa dell'aggiornamento delle prime e ultime righe della matrice, tenendo conto delle condizioni ai bordi. Per fare ciò, considera il numero di celle vive adiacenti a ciascuna cella nelle prime e ultime righe. A seconda della posizione, vengono valutate le celle adiacenti nelle righe superiori, inferiori e corrispondenti, al fine di calcolare il numero di celle vive (n_alive) per ciascuna cella.

La funzione gestisce anche il caso in cui la matrice abbia solo una riga, trattando questa riga come le prime righe con la riga superiore e inferiore vuote.

Nel ciclo principale, per ciascuna cella nelle prime righe (first_row) e nelle ultime righe (last_row), calcola il numero di celle vive adiacenti utilizzando la funzione checkStatus. Quindi, applica la funzione update_matrix per aggiornare la matrice next_era_matrix basandosi sui valori calcolati.

Parametri

Nome	Descrizione
matrix	Matrice dell' era corrente
next_era_matrix	Matrice della prossima era
n_row	Numero di righe della matrice
n_cols	Numero di colonne della matrice
rank	Rank del processo
top_row	Riga superiore
bottom_row	Riga inferiore

```
void exec_last_first_rows(char * matrix, char *next_era_matrix, int n_row, int n_cols, int

int n_alive =0;
int first_row =0;
if(n_row>1){
    int last_row = n_row-1;
    int n_alive_last_row=0;
    for (int j = 0; j < n_cols; j++){
        int prev_col = (j == 0) ? n_cols - 1 : j - 1;
        int next_col = (j + 1) % n_cols;
        n_alive=0;
        n_alive_last_row=0;

        n_alive += checkStatus(matrix[(first_row)*n_cols+prev_col]);
        n_alive += checkStatus(matrix[(first_row)*n_cols+j]);
        n_alive += checkStatus(matrix[(first_row)*n_cols+next_col]);

        n_alive += checkStatus(matrix[(1)*n_cols+prev_col]);
        n_alive += checkStatus(matrix[(1)*n_cols+j]);
        n_alive += checkStatus(matrix[(1)*n_cols+next_col]);

        n_alive += checkStatus(bottom_row[prev_col]);
        n_alive += checkStatus(bottom_row[j]);
        n_alive += checkStatus(bottom_row[next_col]);

        n_alive_last_row += checkStatus(top_row[prev_col]);
        n_alive_last_row += checkStatus(top_row[j]);
        n_alive_last_row += checkStatus(top_row[next_col]);
    }
}
```

```

n_alive_last_row += checkStatus(matrix[(last_row)*n_cols+prev_col]);
n_alive_last_row += checkStatus(matrix[(last_row)*n_cols+j]);
n_alive_last_row += checkStatus(matrix[(last_row)*n_cols+next_col]);

n_alive_last_row += checkStatus(matrix[(last_row-1)*n_cols+prev_col]);
n_alive_last_row += checkStatus(matrix[(last_row-1)*n_cols+j]);
n_alive_last_row += checkStatus(matrix[(last_row-1)*n_cols+next_col]);
update_matrix(next_era_matrix,n_alive,n_cols,first_row,j,matrix[first_row*n_cols+j]);

update_matrix(next_era_matrix,n_alive_last_row,n_cols,last_row,j,matrix[last_row*n_cols+
}
} else{
for (int j = 0; j < n_cols; j++){
n_alive=0;
int prev_col = (j == 0) ? n_cols - 1 : j - 1;
int next_col = (j + 1) % n_cols;

n_alive += checkStatus(matrix[first_row*n_cols+prev_col]);
n_alive += checkStatus(matrix[first_row*n_cols+j]);
n_alive += checkStatus(matrix[first_row*n_cols+next_col]);

n_alive += checkStatus(top_row[prev_col]);
n_alive += checkStatus(top_row[j]);
n_alive += checkStatus(top_row[next_col]);

n_alive += checkStatus(bottom_row[prev_col]);
n_alive += checkStatus(bottom_row[j]);
n_alive += checkStatus(bottom_row[next_col]);
update_matrix(next_era_matrix,n_alive,n_cols,first_row,j,matrix[first_row*n_cols+j]);
}
}
}

```

dead_or_alive

Questa funzione utilizza la funzione rand() per generare un numero casuale. In base al numero casuale generato, decide se la cella dovrebbe essere viva ('A') o morta ('D') e restituisce il carattere corrispondente.

Parametri

X

```
char dead_or_alive(){
    int random_number = rand() % 2;
    if (random_number == 0) {
        return 'D';
    } else {
        return 'A';
    }
}
```

take_args

Questa funzione analizza gli argomenti passati da riga di comando e assegna i valori corrispondenti alle variabili N(numero di righe), M(numero di colonne), e I(numero di iterazioni).

Parametri

Nome	Descrizione
argc	Numero di argomenti passati da riga di comando
argv[]	Array contente gli argomenti passati da riga di comando
N	Variabile per immagazzinare il valore da riga di comando
M	Variabile per immagazzinare il valore da riga di comando
I	Variabile per immagazzinare il valore da riga di comando

```
void take_args(int argc, char *argv[], int *N, int *M, int *I){
    for (int i = 1; i < argc; i++) {
        if (strstr(argv[i], "--n=") == argv[i]) {
            *N = atoi(argv[i] + 4);
        } else if (strstr(argv[i], "--m=") == argv[i]) {
            *M = atoi(argv[i] + 4);
        } else if (strstr(argv[i], "--i=") == argv[i]) {
            *I = atoi(argv[i] + 4);
        }
    }
}
```

MAIN

Analizziamo il flusso di eventi:

```
int main(int argc, char *argv[]) {
    int N = -1;
    int M = -1;
    int I = -1;
    int rank, size;
    int row_x_process, remainings, offset;
    double start_time;
    double end_time;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    take_args(argc, argv, &N, &M, &I);

    if (N == -1 M == -1 I == -1) {
        if (rank==0){
            printf("You must use -M -N -I.\n");
        }
        MPI_Finalize();
        return 0;
    }
}
```

In questa parte iniziale del codice inizializza l'ambiente MPI, ottiene il rank e il numero totale di processi, chiama la funzione `take_args` per ottenere i parametri di input e controlla se sono stati specificati correttamente. Se i parametri non sono stati specificati correttamente, stampa un messaggio di errore e termina l'ambiente MPI.

```
char* matrix = malloc(N * M * sizeof(char));

unsigned int seed = (unsigned int)time(NULL);

if (rank == 0) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            matrix[i * M + j] = dead_or_alive();
        }
    }
}
```

Questo blocco di codice genera il seme randomico per generare i valori e assicura che solo il processo con rank 0 popoli la matrice iniziale tramite la funzione `dead_or_alive`


```

int* sendcounts = malloc(size * sizeof(int));
int* displs = malloc(size * sizeof(int));

row_x_process = N / size;

remainings = N % size;

offset=0;

for (int i = 0; i < size; i++){
    if(i < remainings){
        sendcounts[i]= (row_x_process + 1)*M;
    }else{
        sendcounts[i]= (row_x_process*M);
    }
    displs[i] = offset;
    offset += sendcounts[i];
}

```

In questa parte del codice vengono calcolate le divisioni delle righe da assegnare a ciascun processo durante la fase di distribuzione dei dati. La variabile `row_x_process` tiene conto del numero di righe che ciascun processo deve ricevere, ed è ottenuta dividendo il numero totale di righe per il numero totale di processi nel comunicatore MPI.

La variabile `remainings` viene utilizzata per verificare se il numero totale di righe è divisibile in modo uniforme tra tutti i processi. Se ci sono righe rimanenti che non possono essere distribuite equamente, allora il valore di `remainings` sarà maggiore di zero.

Successivamente, l'array `sendcounts` viene riempito. Questo array specifica quanti elementi (in termini di elementi della matrice, non di byte) devono essere inviati a ciascun processo durante la fase di scattering. Nel caso in cui le righe possano essere distribuite equamente, a ciascun processo viene assegnato un numero di righe pari a `row_x_process`. Altrimenti, i primi `remainings` processi riceveranno un numero di righe maggiorato di uno rispetto a `row_x_process`.

La variabile `displs` rappresenta gli spostamenti iniziali per ciascun processo durante il ricevimento dei dati. Questi spostamenti indicano da quale posizione iniziare a copiare i dati nella matrice di origine. Gli spostamenti sono calcolati in base ai valori di `sendcounts` per assicurarsi che ogni processo riceva i dati corretti.

ESEMPIO

In questo esempio, viene utilizzata una matrice 10x5 tra 4 processi.

- `N` (numero totale di righe nella matrice) = 10
- `M` (numero totale di colonne nella matrice) = 5
- `size` (numero totale di processi MPI) = 4

Ora possiamo calcolare la distribuzione delle righe tra i processi:

1. $\text{row_x_process} = N / \text{size} = 10 / 4 = 2$ Quindi ogni processo dovrebbe ricevere blocchi di 2 righe.
2. $\text{remainings} = N \% \text{size} = 10 \% 4 = 2$ Ci sono 2 righe rimanenti che dovrebbero essere distribuite tra i processi.
3. Calcoliamo ora la suddivisione delle righe e degli spiazamenti:
 - o Processo 0: $\text{sendcounts}[0] = (\text{row_x_process} + 1) * M = (2 + 1) * 5 = 15$
 - o Processo 1: $\text{sendcounts}[1] = (\text{row_x_process}) * M = 2 * 5 = 10$
 - o Processo 2: $\text{sendcounts}[2] = (\text{row_x_process}) * M = 2 * 5 = 10$
 - o Processo 3: $\text{sendcounts}[3] = (\text{row_x_process}) * M = 2 * 5 = 10$

Gli spiazamenti (`displs`) saranno:

- o Processo 0: $\text{displs}[0] = 0$
- o Processo 1: $\text{displs}[1] = 15$
- o Processo 2: $\text{displs}[2] = 25$
- o Processo 3: $\text{displs}[3] = 35$

4. Ora calcoliamo il numero di righe `n_row` per ciascun processo:

- o Processo 0: $n_row = \text{sendcounts}[0] / M = 15 / 5 = 3$
- o Processo 1: $n_row = \text{sendcounts}[1] / M = 10 / 5 = 2$
- o Processo 2: $n_row = \text{sendcounts}[2] / M = 10 / 5 = 2$
- o Processo 3: $n_row = \text{sendcounts}[3] / M = 10 / 5 = 2$

```
MPI_Scatterv(matrix,sendcounts,displs,MPI_CHAR,local_matrix,recvcount,MPI_CHAR,0,MPI_COMM_WORL
MPI_Barrier(MPI_COMM_WORLD);
```

La funzione `MPI_Scatterv` viene utilizzata per distribuire dati da un processo "root" agli altri processi in un comunicatore MPI. I parametri `sendcounts` e `displs` specificano quanti dati devono essere inviati a ciascun processo destinatario e da quale posizione iniziale devono essere presi i dati nel buffer di invio (`matrix`).

La funzione `MPI_Barrier` viene utilizzata per sincronizzare i processi all'interno di un comunicatore MPI

```

char* next_era_matrix = malloc(n_row * M * sizeof(char));
char* top_row=malloc(M * sizeof(char));
char* bottom_row=malloc(M * sizeof(char));

MPI_Request req_bot, res_bot, req_top, res_top;
MPI_Status stat;

int completed_b;
int completed_t;

int pred, next;

pred = (rank == 0) ? (size - 1) : (rank - 1);
next = (rank == (size - 1)) ? 0 : (rank + 1);

```

Qui vengono inizializzate delle variabili che serviranno per lo scambio delle righe di ciascun processo.

```

for (int iter = 0; iter < I; iter++){
    MPI_Isend(local_matrix,M,MPI_CHAR,pred,0,MPI_COMM_WORLD,&req_top);
    MPI_Irecv(top_row,M,MPI_CHAR,next,0,MPI_COMM_WORLD, &res_top);
    MPI_Isend(local_matrix+((n_row-1)*M),M,MPI_CHAR,next,0,MPI_COMM_WORLD,&req_bot);
    MPI_Irecv(bottom_row,M,MPI_CHAR,pred,0,MPI_COMM_WORLD, &res_bot);

    if(n_row>2){
        exec_middle_row(local_matrix,next_era_matrix,n_row,M,rank);
    }

    MPI_Test(&res_bot, &completed_b, MPI_STATUS_IGNORE);
    MPI_Test(&res_top, &completed_t, MPI_STATUS_IGNORE);

    if (!completed_b !completed_t) {
        MPI_Wait(&res_bot,&stat);
        MPI_Wait(&res_top,&stat);
    }

    exec_last_first_rows(local_matrix,next_era_matrix,n_row,M,rank,top_row,bottom_row);
    memcpy(local_matrix, next_era_matrix, n_row * M * sizeof(char));
}

```

In questo pezzo di codice viene eseguito il ciclo che gestisce le iterazioni e la logica di gioco. Ad ogni iterazione i processi eseguono le operazioni di invio e ricezione asincrone tra processi vicini per la comunicazione delle righe superiori e inferiori delle matrici locali.

Viene eseguito un controllo per vedere se i processi possono iniziare ad operare sulle righe intermedie che non necessitano della prima ed ultima riga per l' aggiornamento.

Successivamente viene verificata se l' operazione di invio e ricezione asincrona è stata completata, in caso positivo si prosegue altrimenti l' esecuzione rimane in attesa del completamento.

In fine, viene chiamata la funzione che gestisce l'aggiornamento delle linee di bordo ed eseguita la riga di codice utilizza `memcpy` per copiare i dati dalla matrice `next_era_matrix` alla matrice `local_matrix`, che rappresenta la matrice delle righe attuali che il processo sta gestendo, in modo che i nuovi dati possano essere utilizzati come input per il calcolo successivo.

```
MPI_Gatherv(next_era_matrix, sendcounts[rank], MPI_CHAR, matrix, sendcounts, displs, MPI_CHAR, 0, MPI_
MPI_Barrier(MPI_COMM_WORLD);
```

La funzione `MPI_Gatherv` raccoglie i dati dalla matrice `next_era_matrix` di ciascun processo e li riunisce nella matrice `matrix` nel processo con rank 0. Successivamente, viene utilizzata la funzione `MPI_Barrier` per sincronizzare i processi.

```
if (rank==0){
    end_time = MPI_Wtime();
    printf("Time taked = %f\n", end_time - start_time);
}

free(sendcounts);
free(displs);
free(local_matrix);
free(next_era_matrix);
free(matrix);
free(bottom_row);
free(top_row);

MPI_Finalize();
return 0;
}
```

Questo blocco finale del codice oltre a dare in output il tempo di esecuzione, esegue alcune operazioni finali e pulizie prima della chiusura del programma MPI.

-CORRETTEZZA-

Per valutare la correttezza della soluzione, è eseguito un test per osservare il comportamento quando il numero di processi viene incrementato utilizzando la stessa matrice iniziale (nel codice viene

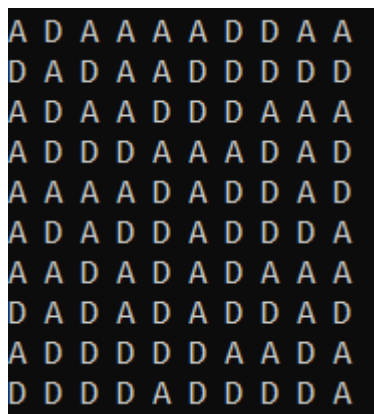
rimosso il seed randomico in modo da generare la stessa matrice).

L'obiettivo è osservare se le matrici di output sono le stesse.

Per il test viene utilizzata questa configurazione:

- N: 10
- M: 10
- l: 1
- np: 1 - 5 - 10

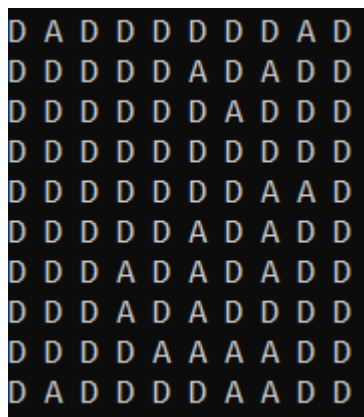
Matrice iniziale:



A	D	A	A	A	A	D	D	A	A
D	A	D	A	A	D	D	D	D	D
A	D	A	A	D	D	D	A	A	A
A	D	D	D	A	A	A	D	A	D
A	A	A	A	D	A	D	D	A	D
A	D	A	D	D	A	D	D	D	A
A	A	D	A	D	A	D	A	A	A
D	A	D	A	D	A	D	D	A	D
A	D	D	D	D	D	A	A	D	A
D	D	D	D	A	D	D	D	D	A

Matrice con 1 processo:

```
mpirun -np 1 --allow-run-as-root --n=10 --m=10 --i=1
```



D	A	D	D	D	D	D	D	A	D
D	D	D	D	D	A	D	A	D	D
D	D	D	D	D	D	A	D	D	D
D	D	D	D	D	D	D	D	D	D
D	D	D	D	D	D	D	A	A	D
D	D	D	D	D	A	D	A	D	D
D	D	D	A	D	A	D	A	D	D
D	D	D	A	D	A	D	D	D	D
D	D	D	D	A	A	A	A	D	D
D	A	D	D	D	D	A	A	D	D

Matrice con 5 processi:

```
mpirun -np 5 --allow-run-as-root --n=10 --m=10 --i=1
```

```
D A D D D D D D A D
D D D D D A D A D D
D D D D D D D A D D
D D D D D D D D D D
D D D D D D D A A D
D D D D D A D A D D
D D D A D A D A D D
D D D A D A D D D D
D D D D A A A A D D
D A D D D D A A D D
```

Matrice con 10 processi:

```
mpirun -np 10 --allow-run-as-root --n=10 --m=10 --i=1
```

```
D A D D D D D D A D
D D D D D A D A D D
D D D D D D D A D D
D D D D D D D D D D
D D D D D D D A A D
D D D D D A D A D D
D D D A D A D A D D
D D D A D A D D D D
D D D D A A A A D D
D A D D D D A A D D
```

Da come si può vedere aumentando il numero di processi ed utilizzando la stessa matrice le matrici di output risultano uguali.

-BENCHMARKS-

la soluzione viene presentata in termini di scalabilità forte e scalabilità debole utilizzando un cluster su Google Cloud composto da 6 istanze (e2-standard-4) ciascuna con 4 vCPU 2 Core e 16 GB RAM us-central-1a ed 2 istanze (e2-standard-4) con 4vCPU 2Core e 16GB RAM us-east-1b.

SCALABILITÀ FORTE

Nel contesto dell'analisi di scalabilità forte, sono state valutate cinque matrici con dimensioni fisse, variando il numero di processi da 1 a 32. Per mantenere una base di confronto uniforme, è stata utilizzata un'impostazione costante di 100 iterazioni.

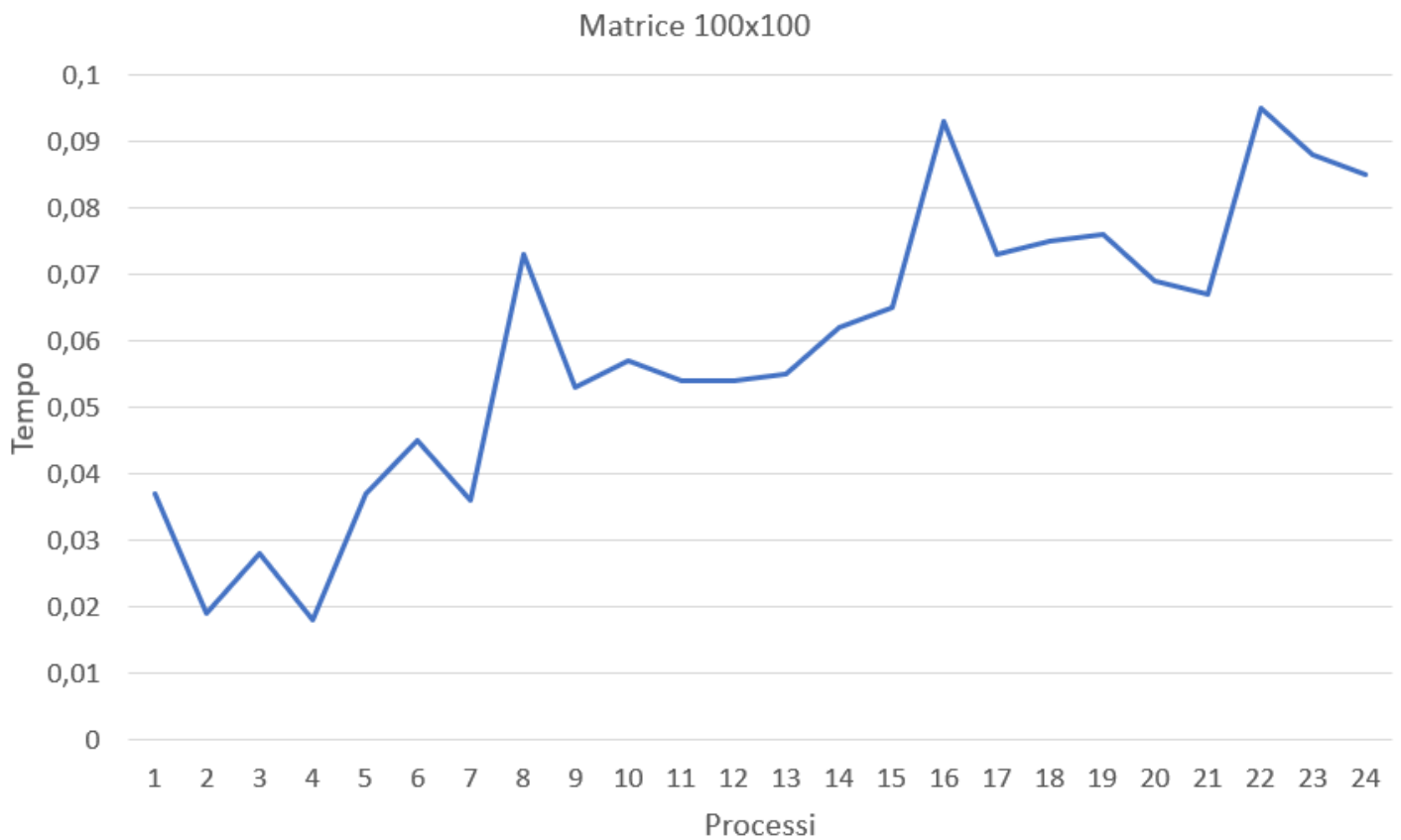
Dimensioni testate:

- 100x100
- 500x500
- 1000x1000
- 1500x1500
- 2000x2000

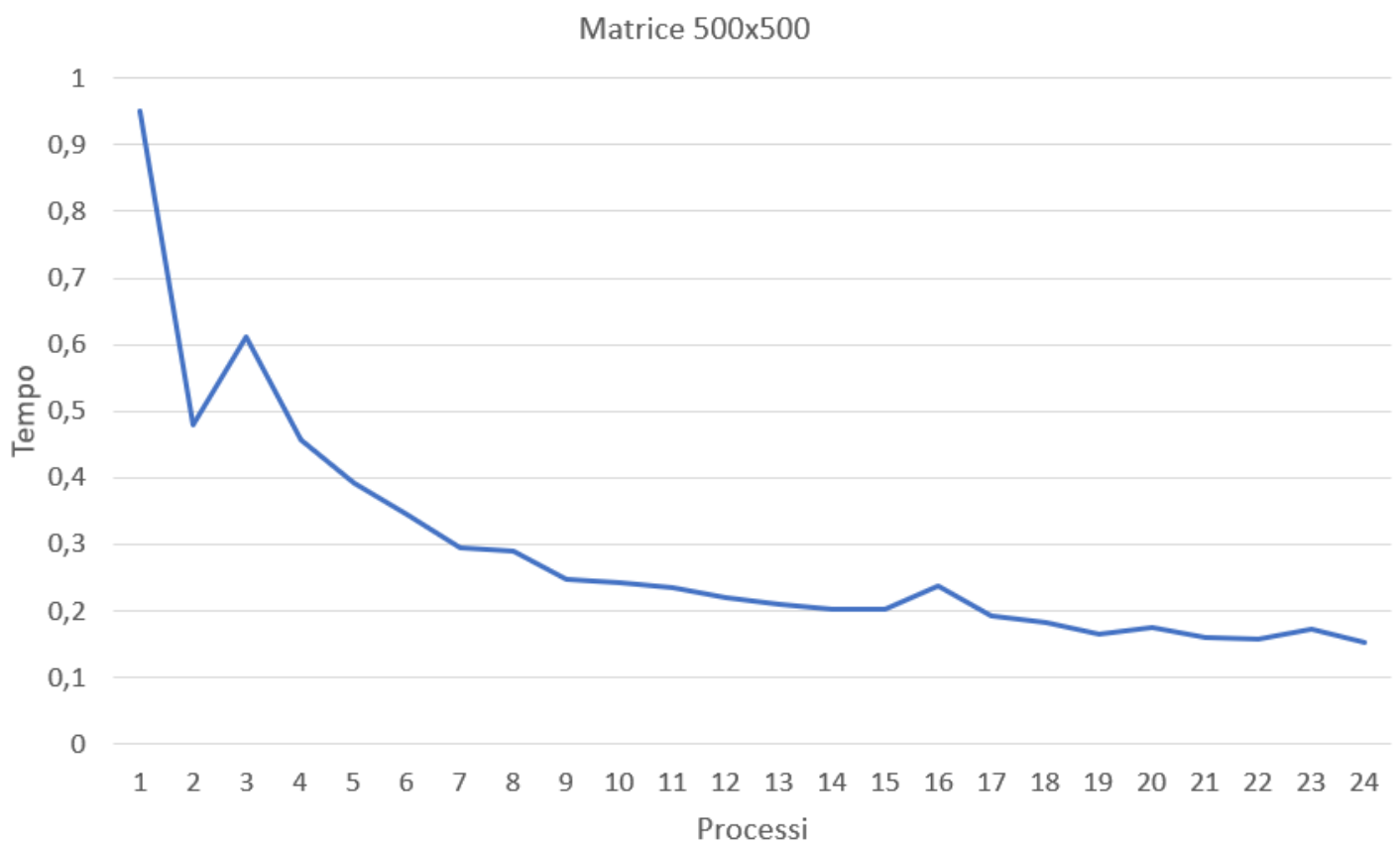
Note

Nella presentazione dei risultati, sono stati inclusi solamente i primi 24 processi. Questa scelta è motivata dal fatto che sono state utilizzate 8 istanze, di cui 2 operano su una zona diversa. L'inclusione di queste istanze con zone diverse ha comportato un considerevole aumento nei tempi di esecuzione complessivi (da come si può notare nelle tabelle). Pertanto, al fine di mantenere un'analisi chiara e rappresentativa, è stata presa la decisione di escluderle dalla visualizzazione grafica dei risultati.

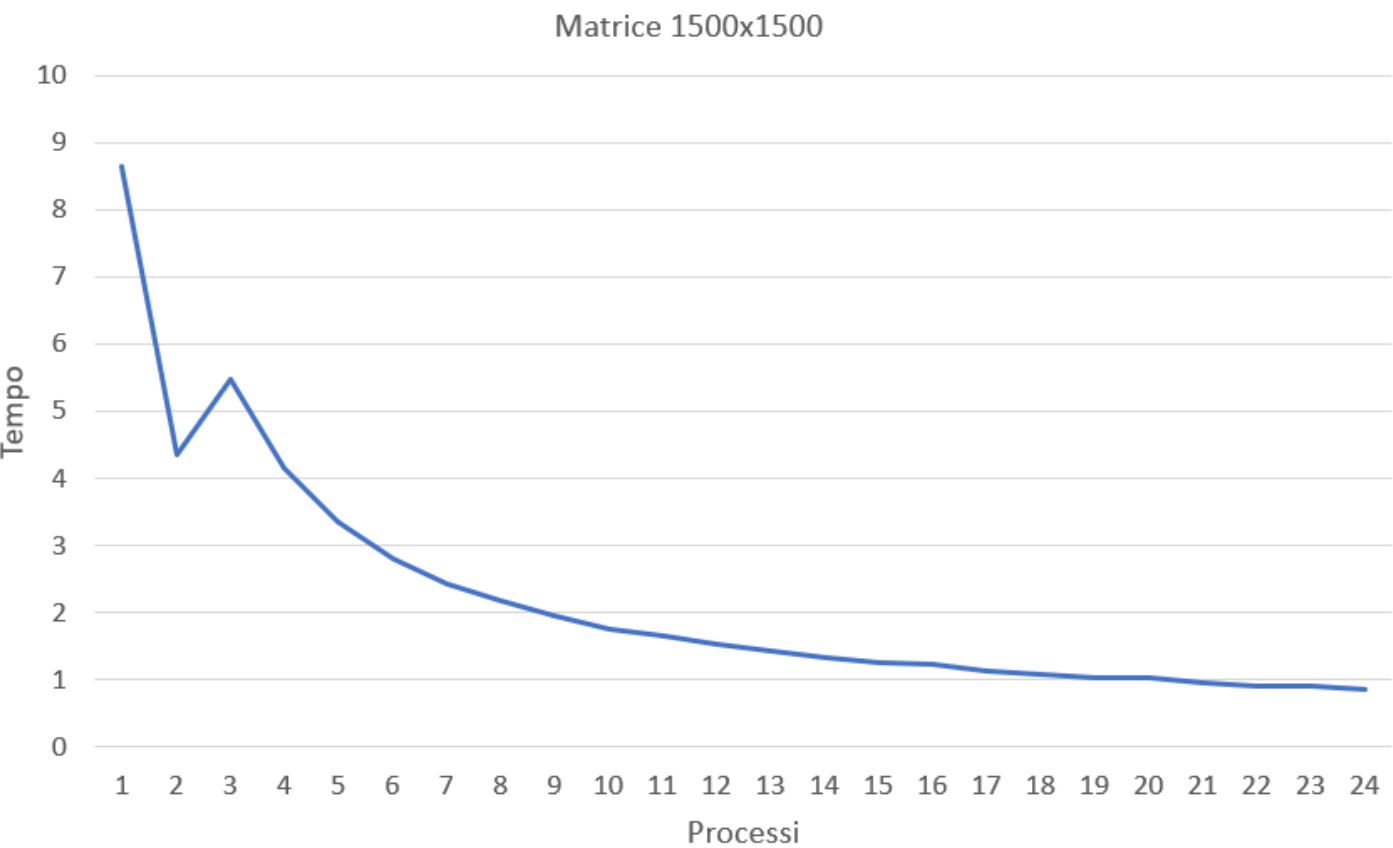
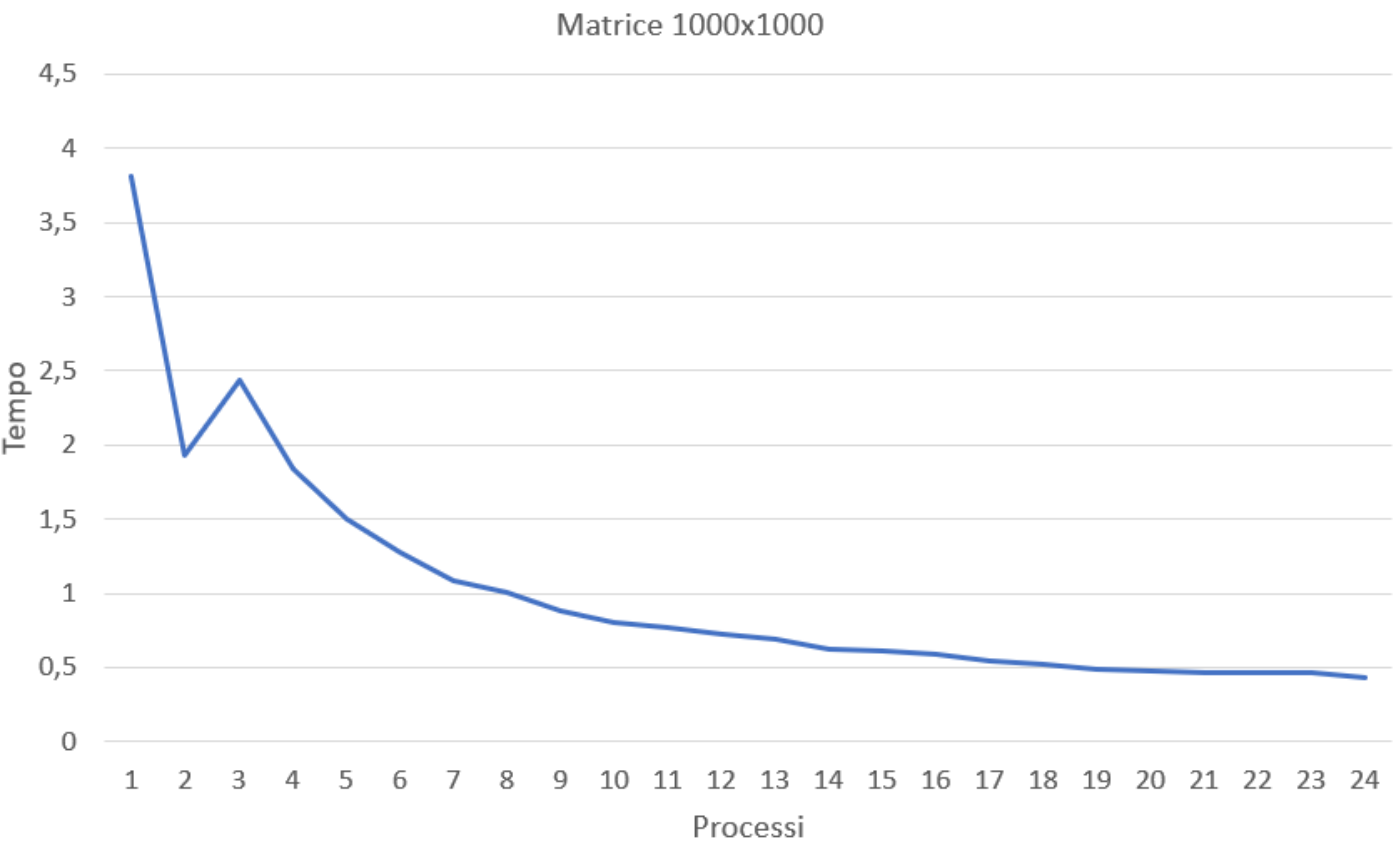
Risultati:



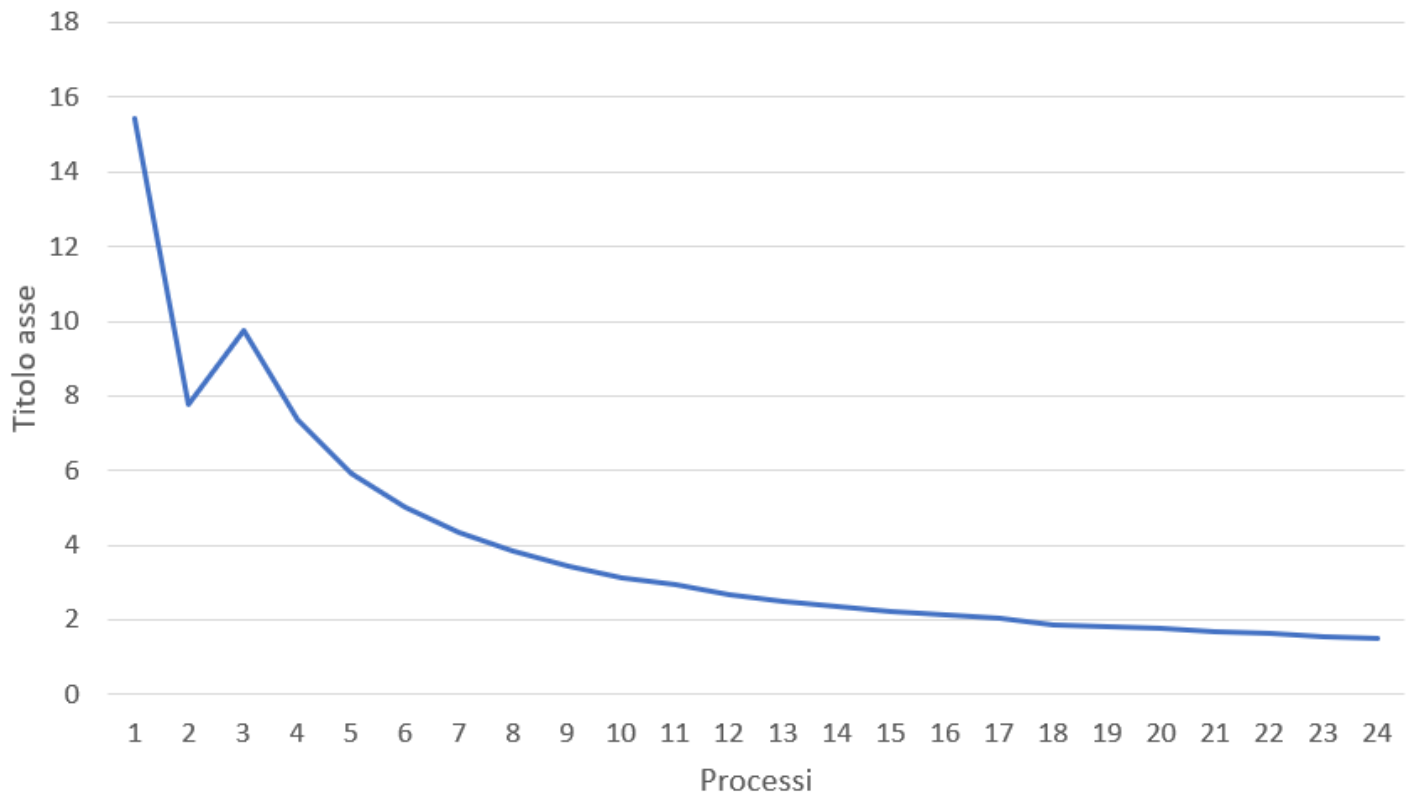
L'analisi dei risultati e del grafico evidenzia alcune tendenze interessanti nella soluzione proposta per la matrice 100x100. Si nota che i tempi di esecuzione non sono uniformi, e questo potrebbe indicare una scarsa scalabilità della soluzione in presenza di matrici di piccola taglia.



Tuttavia, i miglioramenti diventano evidenti già con una matrice di dimensioni 500x500. Aumentando il numero dei processi, si nota una riduzione del tempo di esecuzione, seppur di entità limitata.



Matrice 2000x2000



Proseguendo con l'aumento della dimensione della matrice, si osserva che, nonostante un piccolo aumento nel tempo di esecuzione con 3 processi, il tempo globale continua a diminuire gradualmente. La tendenza generale di miglioramento suggerisce una maggiore scalabilità del programma all'aumentare delle dimensioni della matrice e dei processi coinvolti.

Tabella dei valori

N Processi	100x100	500x500	1000x1000	1500x1500	2000x2000
1	0.037	0.952	3.81	8.642	15.414
2	0.019	0.48	1.925	4.357	7.764
3	0.028	0.612	2.442	5.483	9.741
4	0.018	0.458	1.841	4.147	7.381
5	0.037	0.392	1.503	3.351	5.941
6	0.045	0.345	1.280	2.816	5.041
7	0.036	0.295	1.090	2.44	4.349
8	0.073	0.289	1.005	2.186	3.857
9	0.053	0.249	0.881	1.952	3.432
10	0.057	0.244	0.807	1.766	3.144

N Processi	100x100	500x500	1000x1000	1500x1500	2000x2000
11	0.054	0.235	0.771	1.648	2.927
12	0.054	0.220	0.724	1.542	2.685
13	0.055	0.210	0.692	1.427	2.509
14	0.062	0.203	0.621	1.335	2.366
15	0.065	0.202	0.610	1.266	2.240
16	0.093	0.237	0.591	1.238	2.138
17	0.073	0.192	0.542	1.140	2.023
18	0.075	0.183	0.521	1.078	1.886
19	0.076	0.166	0.493	1.029	1.807
20	0.069	0.176	0.476	1.024	1.756
21	0.067	0.160	0.465	0.964	1.683
22	0.095	0.157	0.468	0.920	1.646
23	0.088	0.172	0.461	0.900	1.557
24	0.085	0.153	0.430	0.868	1.490
25	2.057	2.078	2.265	2.388	2.524
26	2.119	2.163	2.275	2.514	2.756
27	2.222	2.162	2.273	2.793	3.018
28	2.225	2.290	2.304	2.967	3.292
29	2.401	2.292	2.346	3.081	3.523
30	2.408	2.403	2.473	3.330	3.748
31	2.425	2.443	2.497	3.435	4.095
32	2.34	2.417	2.52	3.554	4.111

SCALABILITÀ DEBOLE

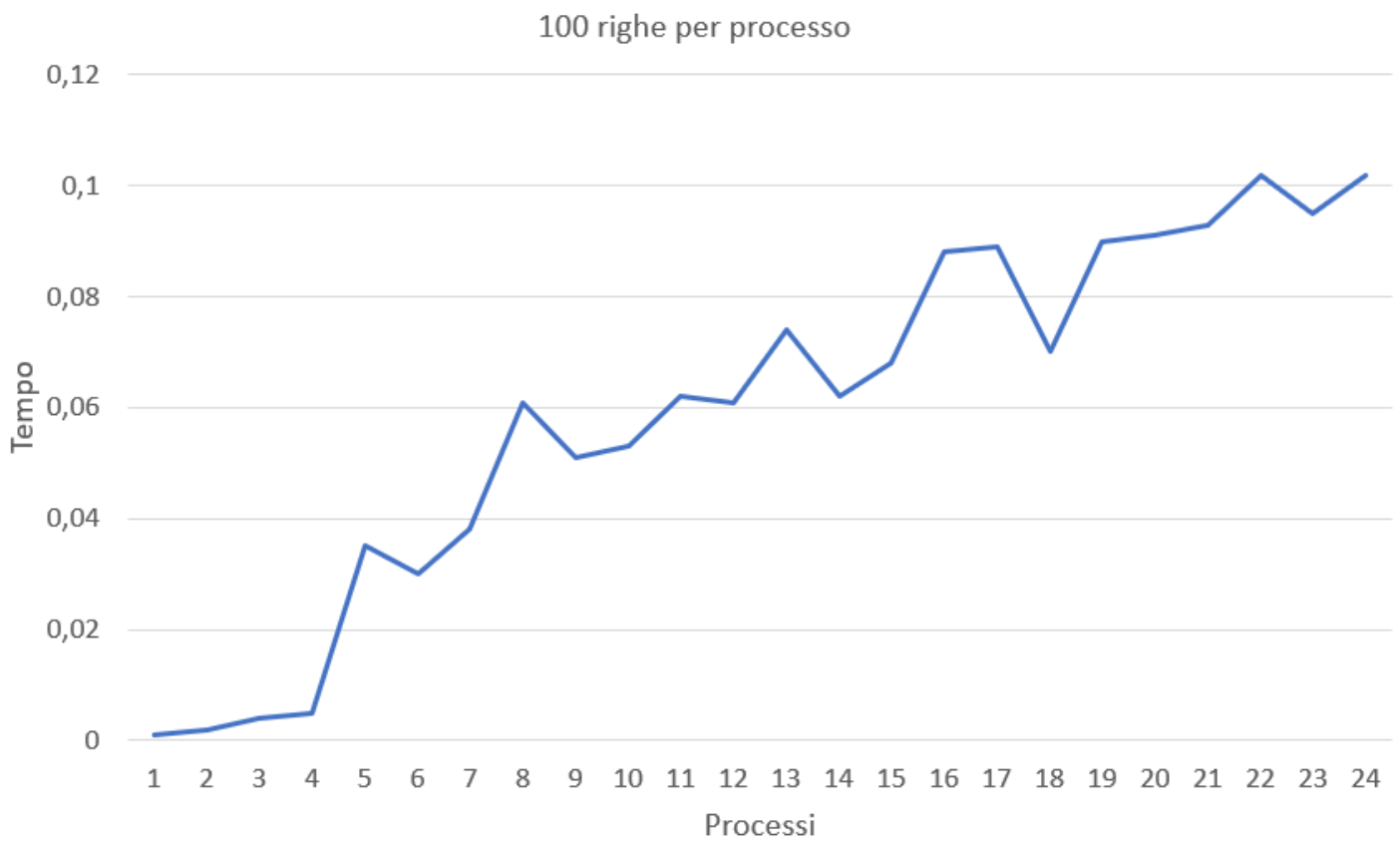
Nell'analisi della scalabilità debole, è stato adottato un approccio in cui il numero di righe per processo è variato in modo tale che, all'aumentare del numero di processi, il numero totale di righe aumentasse proporzionalmente (es. per 1 processo utilizzo una matrice 100x100 per 2 processi

200x100 per 3 processi 300x100 ecc..). Questo è stato realizzato mantenendo costante il numero di iterazioni e il numero di colonne della matrice.

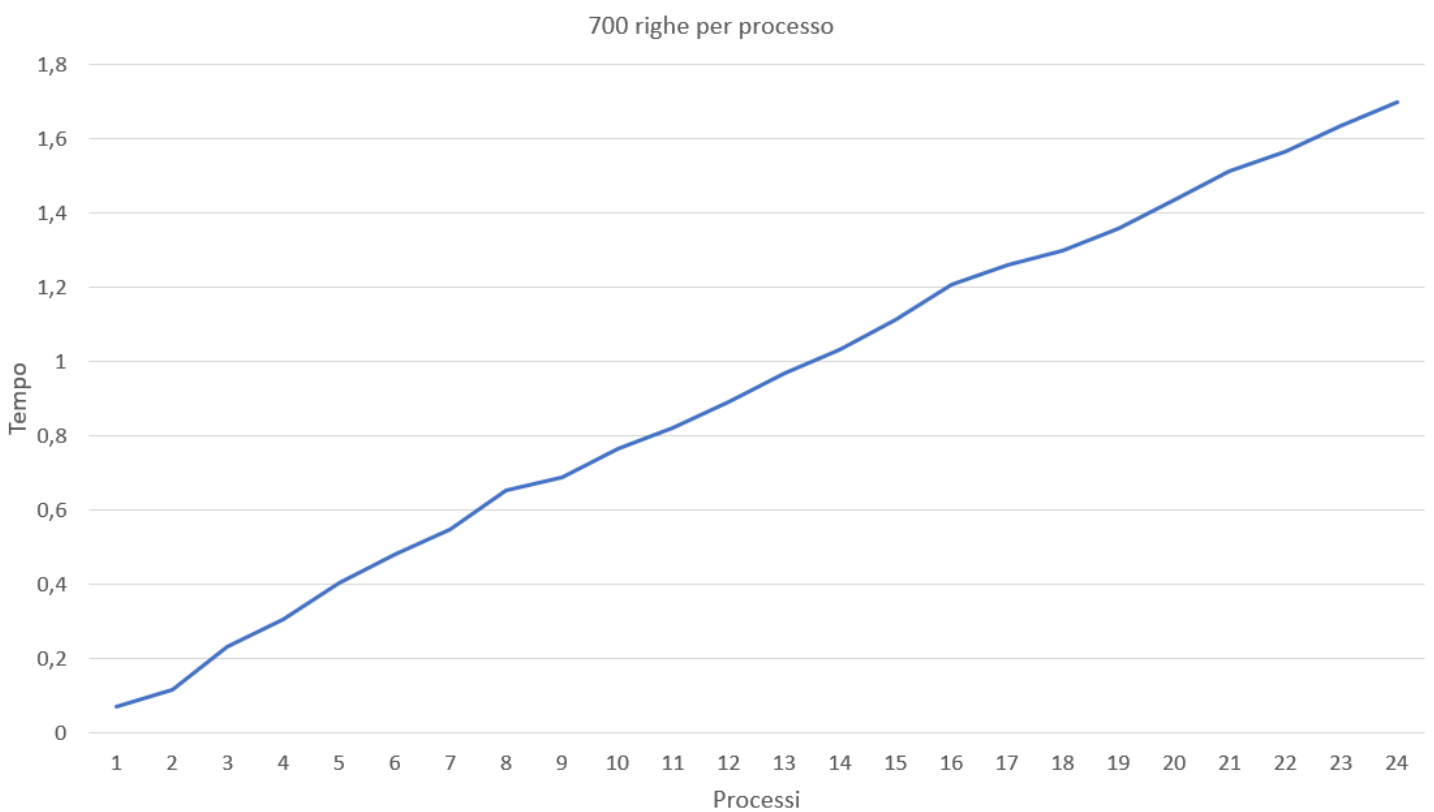
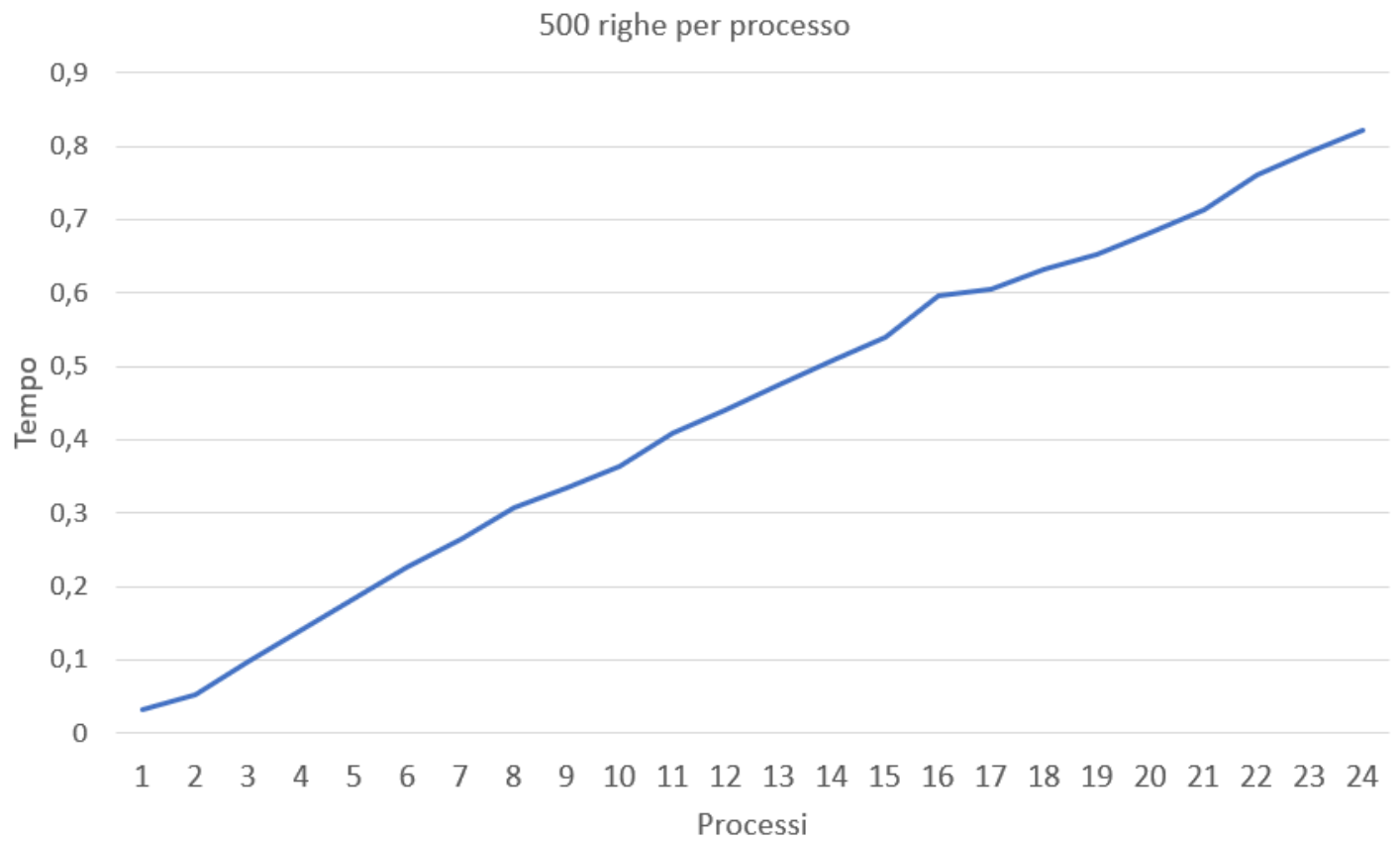
In altre parole, l'obiettivo era quello di distribuire il carico di lavoro in modo equo tra i processi mentre si aumentava il parallelismo attraverso l'aumento del numero di processi, tenendo costanti le altre variabili chiave.

Dimensione testate:

- 100 righe per processo
- 500 righe per processo
- 700 righe per processo



Nel caso della scalabilità debole, anche utilizzando una matrice di dimensioni 100x100, possiamo ancora osservare un comportamento non omogeneo. Ciò significa che l'aumento del numero di processi non ha portato a un miglioramento uniforme delle prestazioni, ma piuttosto a fluttuazioni nei tempi di esecuzione.



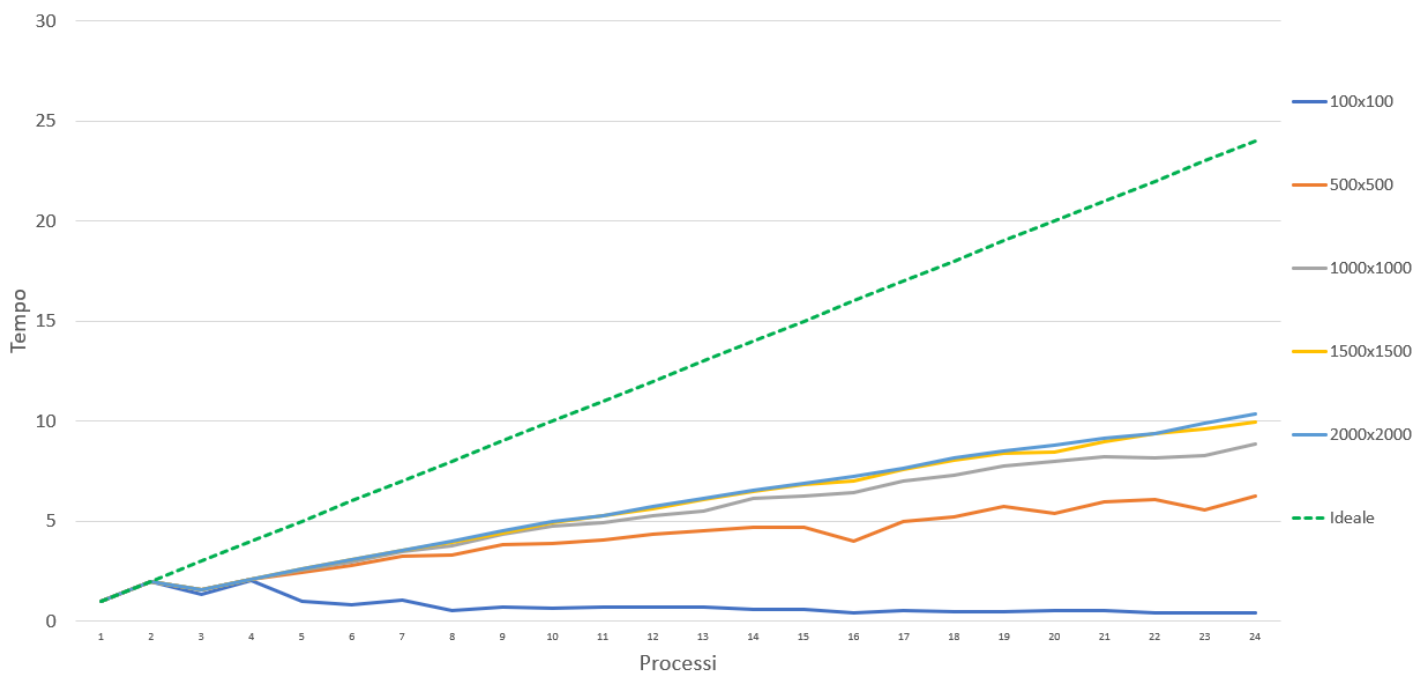
I risultati dei test con 500 e 700 righe per processo evidenziano chiaramente che l'adeguamento delle dimensioni del problema ha avuto un impatto positivo sulla scalabilità del programma MPI.

Tabella dei valori

N Processi	100	500	700
1	0.001	0.031	0.07
2	0.002	0.052	0.116
3	0.004	0.097	0.234
4	0.005	0.141	0.306
5	0.035	0.183	0.406
6	0.030	0.225	0.481
7	0.038	0.265	0.548
8	0.061	0.308	0.655
9	0.051	0.335	0.690
10	0.053	0.363	0.766
11	0.062	0.408	0.823
12	0.061	0.440	0.894
13	0.074	0.474	0.971
14	0.062	0.509	1.033
15	0.068	0.539	1.114
16	0.088	0.596	1.208
17	0.089	0.606	1.261
18	0.070	0.632	1.301
19	0.090	0.652	1.358
20	0.091	0.682	1.435
21	0.093	0.714	1.513
22	0.102	0.761	1.566
23	0.095	0.793	1.635
24	0.102	0.822	1.700
25	0.872	1.529	2.500
26	0.954	1.874	2.780
27	1.140	2.225	3.405

N Processi	100	500	700
28	1.094	2.704	3.883
29	1.171	2.988	4.361
30	1.285	3.685	4.851
31	1.356	3.361	5.300
32	1.284	3.869	5.556

SPEEDUP



Dall'analisi del grafico, è evidente che lo speedup ottenuto non è ottimale. In particolare, quando si lavora con dimensioni di matrice relativamente piccole, i risultati sono notevolmente scadenti. Tuttavia, man mano che le dimensioni della matrice aumentano, lo speedup migliora in proporzione. Nonostante ciò, rimane comunque distante dall'ottimo auspicato.

-CONCLUSIONI-

Si è osservato che i tempi di esecuzione non sono uniformi per matrici di piccole dimensioni, suggerendo che l'efficienza del programma potrebbe essere influenzata maggiormente dalla comunicazione tra i processi rispetto all'elaborazione locale. Questo potrebbe risultare in un aumento dell'overhead di comunicazione rispetto al reale carico di lavoro eseguito dai singoli processi, compromettendo l'efficienza e la scalabilità del programma.

Già con una matrice delle dimensioni di 500x500, si sono notati miglioramenti nell'efficienza attraverso l'aumento del numero di processi. Anche se modesti, questi miglioramenti indicano che l'incremento dei processi sta iniziando a influenzare positivamente la parallelizzazione.

Continuando ad aumentare la dimensione della matrice, si è rilevato che il tempo globale di esecuzione continua a diminuire gradualmente. Questo comportamento suggerisce che l'overhead di comunicazione sta diventando meno rilevante rispetto all'incremento delle dimensioni della matrice e all'effettivo lavoro eseguito dai processi. Questo indica una maggiore scalabilità del programma all'aumentare delle dimensioni sia della matrice che dei processi coinvolti.

In fine, si è visto che lo speedup ottenuto non è ottimale, specialmente con dimensioni di matrice relativamente piccole. Tuttavia, man mano che le dimensioni crescono, lo speedup migliora in proporzione.