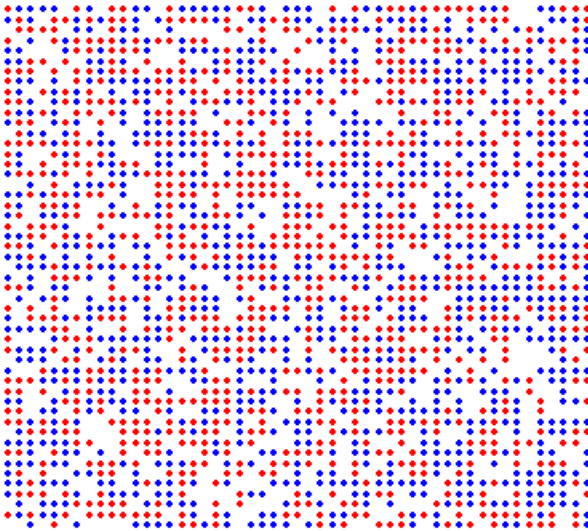


Schelling's model of segregation

Introduzione

A cavallo tra gli anni 60 e 70 del Novecento, l'economista Thomas Schelling, condusse degli studi con cui si proponeva di indagare l'influenza delle preferenze individuali nel determinare la segregazione spaziale; per far questo, Schelling utilizzò un modello a più agenti intelligenti: a interagire nel sistema erano automi cellulari costituiti da pedine di diverso colore su una scacchiera, il cui movimento da una casella all'altra era condizionato, ogni volta, dall' "infelicità" della posizione occupata, a sua volta legato al colore delle pedine più vicine: tali modelli hanno mostrato che è sufficiente che le persone coltivino una blanda preferenza di qualche tipo (ad esempio, etnica, sociale, culturale, ecc.) perché l'effetto di scelte individuali ispirate da tali preferenze debolissime si componga in un fenomeno complessivo di totale segregazione, senza che, nella spiegazione dei fenomeni di separazione in gruppi così nettamente separati, sia possibile distinguere i motivi intenzionali da quelli involontari.

Step 1



Descrizione della soluzione

La soluzione è basata sulla seguente implementazione: inizialmente viene generata la matrice andando ad inserire gli elementi in modo casuale, successivamente la matrice generata viene divisa tra i vari processi che calcolano gli agenti non soddisfatti e scegliendo loro una posizione casuale dove inserirli. Una volta risolta la matrice o finito il numero di iterazioni massimo, i vari processi mandano le proprie porzioni di matrice al rank master che si occuperà poi di stampare la matrice risultante.

Dettagli dell'implementazione

Inizialmente ci sono alcuni parametri modificabili prima di effettuare la compilazione, che permettono di andare a definire la struttura della matrice.

```
#define SIZE 100           //Indica il numero di righe e colonne che la matrice avrà
#define TOLLERANCE 51      //Indica la tolleranza che devono avere gli elementi della matrice
#define PERC_O 50          //Viene indicata la percentuale di elementi 'O' all'interno della matrice
#define PERC_X 100-PERC_O //Viene indicata la percentuale di elementi 'X' all'interno della matrice
#define WHITE_SPACES 30    //Viene indicata la percentuale di caselle vuote all'interno della matrice
#define ITERATIONS 100     //Viene indicato il numero massimo di iterazioni che l'algoritmo esegue
#define RANDOM_MATRIX 1    //Con 0 viene generata una matrice costante, con 1 una matrice casuale
#define PRINT_MATRIX 0     //Con 1 viene stampata la matrice iniziale e finale, con 0 la stampa viene omessa
```

All'interno dell'implementazione abbiamo la definizione di diverse struct:

- Coord: La struttura Coord viene utilizzata per salvare la posizione dei vari elementi

```
typedef struct
{
    int x;
    int y;
}Coord;
```

- MatElements: La struttura MatElements è usata per mantenere tutte le informazioni riguardanti la matrice creata

```
typedef struct {
    int array_border[2][SIZE]; //array per salvare le righe dei processi vicini
    int **array_op; //array per andare a salvare le righe che verranno utilizzate durante la
    computazione da ciascun processo
    Coord *array_empty_coord; //array per salvare le posizioni di tutti gli elementi vuoti presenti
    all'interno della matrice
    Coord *array_local_empty_coord; //array per salvare le posizioni degli elementi vuoti
    appartenenente ad un processo
    int rows_op; //utilizzato per tenere traccia del quantitativo di righe che un processo possiede
    int initial_row; //utilizzato per tenere traccia la posizione della riga iniziale della porzione
    di matrice appartenente ad un processo
    int dim_array_local_empty; //utilizzato per tenere traccia del numero di celle vuote appartenenti
    ad un processo.
} MatElements;
```

- Element: La struttura Element viene utilizzata per tenere traccia degli elementi scambiati dai diversi processori

```
typedef struct
{
    int rank;
    int val;
    int x;
    int y;
    int prev_x;
    int prev_y;
    int changed;
} Element;
```

Sono state definite anche le seguenti funzioni:

Readme

```
int check(int array_border[],int size); //Controlla se l'elemento rispetta i parametri di tolleranza
all'interno della matrice
void swap(int i,int j,int world_rank,int rows,int num_it); //Effettua lo swap dell'elemento all'interno
della matrice
void print_matrix_complete(); //Viene stampata la matrice completa
void print_matrix_op(); //Viene stampata la matrice appartenente al singolo processo
void def_array_border(int world_rank,int world_size); //Permette lo scambio di righe di bordo tra i
vari processi
void find_places(int world_rank, int world_size,int num_it); //Utilizzata per iterare gli elementi
all'interno della matrice, controllando se ogni elemento è soddisfatto
```

Una volta creata la matrice, il rank master calcola le varie righe da inviare ad ogni processo, andando a memorizzare gli elementi inviati e la riga iniziale della porzione di matrice con cui ogni processo opera:

```

int modulo=SIZE%world_size;
int div=SIZE/world_size;
int check=0;
for(int i=0;i<world_size;i++)
{
    if(modulo>0)
    {
        send_counts[i]=SIZE*(div+1);
        if(i==0)
        {
            initial_row[0]=0;
            displs[i]=0;
        }
        else
        {
            displs[i]=displs[i-1]+send_counts[i-1];
            initial_row[i]=displs[i]/SIZE;
        }
        modulo--;
        check=1;
    }
    else
    {
        send_counts[i]=SIZE*div;
        if(i==0)
        {
            initial_row[0]=0;
            displs[i]=0;
        }
        else
        {
            if(check==1)
            {
                displs[i]=displs[i-1]+send_counts[i-1];
                check=0;
            }
            else
            {
                displs[i]=displs[i-1]+send_counts[i-1];
            }
            initial_row[i]=displs[i]/SIZE;
        }
    }
}
}

```

Gli array **displs** e **send_counts** servono per utilizzare la funzione **MPI_Scatterv** che permette di assegnare in modo dinamico gli elementi ai vari processi, in questo caso le varie porzioni della matrice:

```

MPI_Scatterv(array_completo, send_counts, displs, MPI_INT,&sarray.array_op[0][0],
send_counts[world_rank], MPI_INT, 0, MPI_COMM_WORLD);

```

Successivamente viene utilizzata la funzione **def_array_border** che permette ad ogni processo di avere le righe confinanti della propria porzione di matrice, in modo da poter calcolare se un elemento è soddisfatto o meno:

```

void def_array_border(int world_rank,int world_size)
{
    if(world_size > 1)
    {
        if(world_rank == 0)
        {
            MPI_Status status;
            MPI_Sendrecv(&sarray.array_op[sarray.rows_op-1][0], SIZE, MPI_INT, world_rank+1,
0,&sarray.array_border[0][0], SIZE, MPI_INT, world_rank+1,0, MPI_COMM_WORLD, &status);
        }
        else if(world_rank == world_size -1)
        {
            MPI_Status status;
            MPI_Sendrecv(&sarray.array_op[0][0], SIZE, MPI_INT, world_rank-1, 0,&sarray.array_border[0]
[0], SIZE, MPI_INT, world_rank-1,0, MPI_COMM_WORLD, &status);
        }
        else
        {
            MPI_Status status;
            MPI_Sendrecv(&sarray.array_op[0][0], SIZE, MPI_INT, world_rank-1, 0,&sarray.array_border[0]
[0], SIZE, MPI_INT, world_rank-1,0, MPI_COMM_WORLD, &status);
            MPI_Sendrecv(&sarray.array_op[sarray.rows_op-1][0], SIZE, MPI_INT, world_rank+1,
0,&sarray.array_border[1][0], SIZE, MPI_INT, world_rank+1,0, MPI_COMM_WORLD, &status);
        }
    }
}

```

Una volta che ogni processo ha ottenuto la propria porzione della matrice, potrà operare su di essa fino ad un massimo di **ITERATIONS**, viene utilizzata la funzione **find_places** per controllare ogni elemento all'interno della matrice: ogni elemento viene controllato tramite la funzione **check** che restituirà **1** se l'elemento risulta non soddisfatto, **0** altrimenti:

```

int check(int array_border[],int size)
{
    int count=0;
    int count_to_remove=0;

    if(array_border[0]==32)
    {
        return 0;
    }
    for(int i = 1 ; i< size;i++)
    {
        if(array_border[0]==array_border[i])
            count++;
        if(array_border[i]==32)
            count_to_remove++;
    }
    size--;
    int size_final=size-count_to_remove;
    int final_result;
    if(size_final==0)
        final_result=100;
    else
        final_result=(count*100)/size_final;

    if(final_result < TOLLERANCE )
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

Con la funzione **swap** avviene lo spostamento dell'elemento non soddisfatto, questo spostamento può avvenire in due modi:

1. Viene selezionata una casella appartenente alla porzione della matrice del processo
2. Viene selezionata una casella appartenente alla porzione della matrice di un altro processo

```

void swap(int i,int j,int world_rank,int rows,int num_it)
{
    stop=1;
    srand(time(NULL) + num_it + i + j);
    int r=(rand()+num_it*i+j)%numb_white_spaces;
    int check=0;

    int coord_empty_x=sarray.array_empty_coord[r].x;
    int coord_empty_y=sarray.array_empty_coord[r].y;

    int local_i;

    for(int i=0;i<sarray.dim_array_local_empty;i++)
    {
        if(coord_empty_x==sarray.array_local_empty_coord[i].x && coord_empty_y ==

```

```

sarray.array_local_empty_coord[i].y)
{
    check=1;
    local_i=i;

}

}

if(check==1)
{
    sarray.array_empty_coord[r].x=i+sarray.initial_row;
    sarray.array_empty_coord[r].y=j;
    sarray.array_local_empty_coord[local_i].x=i+sarray.initial_row;
    sarray.array_local_empty_coord[local_i].y=j;
    sarray.array_op[coord_empty_x-sarray.initial_row][coord_empty_y]=sarray.array_op[i][j];
    sarray.array_op[i][j]=32;
}
else
{
    int check_elements=0;
    for(int i=0;i< size_elements;i++)
    {
        if(elements[i].x==coord_empty_x && elements[i].y == coord_empty_y)
            check_elements=1;
    }
    if(check_elements==0)
    {
        Element elem;
        elem.changed=0;
        elem.rank=world_rank;
        elem.val=sarray.array_op[i][j];
        elem.prev_x=i+sarray.initial_row;
        elem.prev_y=j;
        elem.x=coord_empty_x;
        elem.y=coord_empty_y;
        size_elements++;
        elements=realloc(elements,sizeof(Element)*size_elements);
        elements[size_elements-1]=elem;
    }
}
}
}

```

Nel caso in cui viene scelta una casella appartenente ad un altro processo, viene istanziata una variabile di tipo `Element` dove vengono salvate tutte le informazioni riguardanti l'elemento da spostare e viene aggiunto all'array **elements**. Alla fine di ogni iterata della porzione della matrice, viene chiamata la funzione **MPI_Allgather** per raccogliere nella variabile **size_all_elements** la dimensione dell'array **elements** di ogni processo:

```
MPI_Allgather(&size_elements, 1, MPI_INT, size_all_elements_array, 1, MPI_INT,MPI_COMM_WORLD);
```

In questo modo è possibile ottenere la grandezza totale di tutti gli elementi che sono soggetti ad un possibile cambiamento di posizione, viene istanziato l'array di `Element` **all_elements** e tramite la funzione **MPI_Allgather** vengono raccolti tutti gli elementi dei vari processi che verranno salvati nell'array **all_elements**:

```

int size_all_elements_array[world_size];

MPI_Allgather(&size_elements, 1, MPI_INT, size_all_elements_array, 1, MPI_INT, MPI_COMM_WORLD);

int size_all_elements=0;
int counts_swap[world_size];
int displacements_swap[world_size];
for(int i=0; i<world_size; i++)
{
    size_all_elements+=size_all_elements_array[i];
    counts_swap[i]=size_all_elements_array[i];
    displacements_swap[i]= (i==0)? 0 : displacements_swap[i-1]+size_all_elements_array[i-1];
}

Element *all_elements=malloc(sizeof(Element)*size_all_elements);

MPI_Allgatherv(elements, size_elements, elementswap, all_elements, counts_swap, displacements_swap,
elementswap, MPI_COMM_WORLD);

```

Ogni processo itera gli elementi presenti nell'array **all_elements** andando a trovare gli elementi che possono essere messi nelle posizioni vuote di ciascun processo. Se l'elemento fa riferimento ad una casella vuota ancora disponibile nella matrice del processo, viene aggiornata la porzione di matrice rappresentata da **sarray.array_op** e l'array contenente gli spazi vuoti locali del processo **sarray.array_local_empty_coord** ed infine viene aggiunto l'elemento nell'array **elements_changed**:


```

Element *elements_changed=malloc(sizeof(Element));
int size_elements_changed=0;
for(int i=0;i<size_all_elements;i++)
{
    int coord_x=all_elements[i].x;
    int coord_y=all_elements[i].y;

    for(int j=0;j<sarray.dim_array_local_empty;j++)
    {
        if(coord_x== sarray.array_local_empty_coord[j].x && coord_y ==
sarray.array_local_empty_coord[j].y)
        {
            all_elements[i].changed=1;

sarray.array_local_empty_coord[j]=sarray.array_local_empty_coord[sarray.dim_array_local_empty-1];
            sarray.dim_array_local_empty--;

sarray.array_local_empty_coord=realloc(sarray.array_local_empty_coord,sizeof(Coord)*sarray.dim_array_local_empty);

            int x=all_elements[i].x;
            sarray.array_op[x-sarray.initial_row][all_elements[i].y]=all_elements[i].val;
            size_elements_changed++;
            elements_changed=realloc(elements_changed,sizeof(Element)*size_elements_changed);
            elements_changed[size_elements_changed-1]=all_elements[i];
        }
    }
}

```

Successivamente sempre con la funzione **MPI_Allgather** vengono raccolte le dimensioni degli elementi che ogni processo deve cambiare e vengono riuniti tutti gli elementi da cambiare con la funzione **MPI_Allgatherv**:

```

MPI_Allgather(&size_elements_changed, 1, MPI_INT, size_all_elements_changed_array, 1,
MPI_INT,MPI_COMM_WORLD);

int size_all_elements_changed=0;
int counts_swap_changed[world_size];
int displacements_swap_changed[world_size];

for(int i=0;i<world_size;i++)
{
    size_all_elements_changed+=size_all_elements_changed_array[i];
    counts_swap_changed[i]=size_all_elements_changed_array[i];
    displacements_swap_changed[i] = (i==0) ? 0 : displacements_swap_changed[i-
1]+size_all_elements_changed_array[i-1];
}

Element* all_elements_changed=malloc(sizeof(Element)*size_all_elements_changed);

if(size_all_elements_changed>0)
    MPI_Allgatherv(elements_changed, size_elements_changed, elementswap, all_elements_changed,
counts_swap_changed, displacements_swap_changed, elementswap, MPI_COMM_WORLD);

for(int i=0; i< size_all_elements_changed;i++)
{
    if(all_elements_changed[i].rank==world_rank)
    {
        sarray.array_op[all_elements_changed[i].prev_x-sarray.initial_row]
[all_elements_changed[i].prev_y]=32;
        sarray.dim_array_local_empty++;

sarray.array_local_empty_coord=realloc(sarray.array_local_empty_coord,sizeof(Coord)*sarray.dim_array_local

        Coord coord;
        coord.x=all_elements_changed[i].prev_x;
        coord.y=all_elements_changed[i].prev_y;
        sarray.array_local_empty_coord[sarray.dim_array_local_empty-1]=coord;

    }
}

```

Una volta aggiornato **array_local_empty_coord** che tiene traccia delle caselle vuote locali all'interno della matrice, si utilizza la funzione **MPI_Allgatherv** per far sì che ogni processo abbia la visione aggiornata di tutte le celle vuote presenti nella matrice:

```

int size_all_elements_local_empty[world_size];

MPI_Allgather(&sarray.dim_array_local_empty, 1, MPI_INT, size_all_elements_local_empty, 1,
MPI_INT,MPI_COMM_WORLD);

int counts_swap_coord[world_size];
int displacements_swap_coord[world_size];

for(int i=0;i<world_size;i++)
{
    counts_swap_coord[i]=size_all_elements_local_empty[i];
    displacements_swap_coord[i] = (i==0)? 0: displacements_swap_coord[i-
1]+size_all_elements_local_empty[i-1];
}

MPI_Allgatherv(sarray.array_local_empty_coord,sarray.dim_array_local_empty,coordswap,sarray.array_empty_co

```

Alla fine delle iterazioni o se non sono stati effettuati più cambiamenti di elementi da ciascun processo, ogni processo invia la sua porzione di matrice rappresentata con **sarray.array_op** al rank master:

```

int send_rows[world_size];

MPI_Allgather(&send_counts[world_rank],1,MPI_INT,send_rows,1,MPI_INT,MPI_COMM_WORLD);

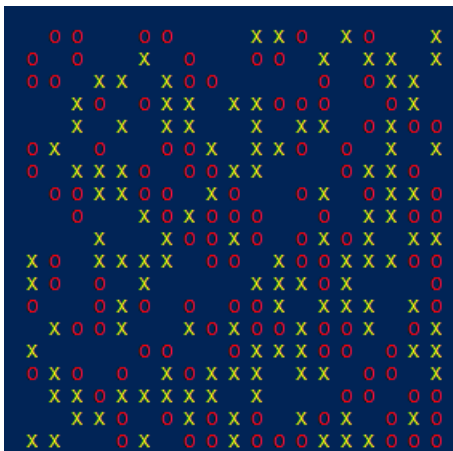
int displs_rows[world_size];
for(int i=0;i<world_size;i++)
    displs_rows[i]= (i==0) ? 0 : displs_rows[i-1]+send_rows[i-1];

MPI_Gatherv(&sarray.array_op[0][0],send_rows[world_rank],MPI_INT,&array_completo[0]
[0],send_rows,displs_rows,MPI_INT,0,MPI_COMM_WORLD);

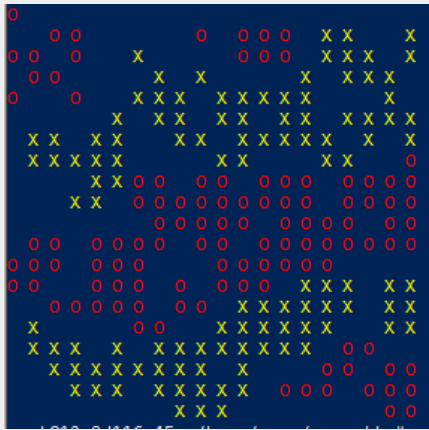
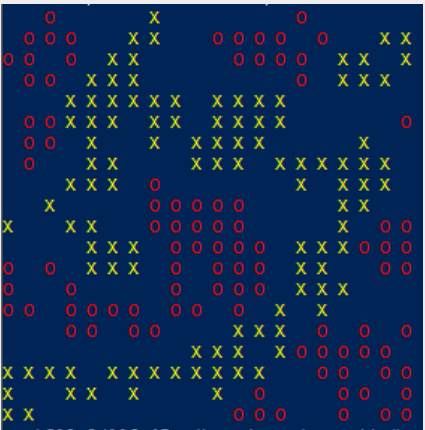
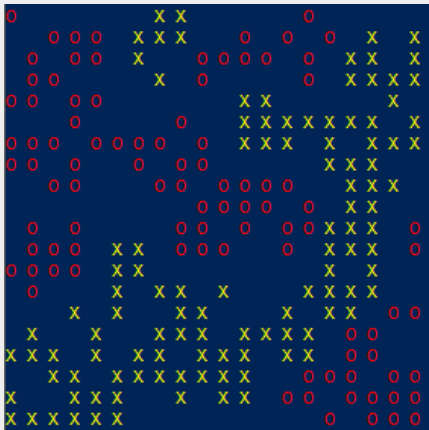
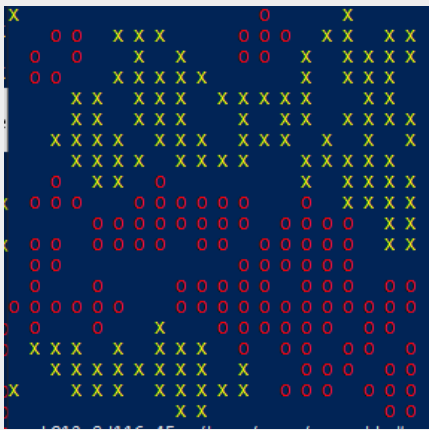
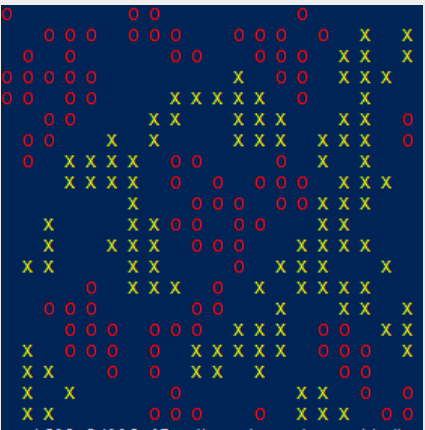
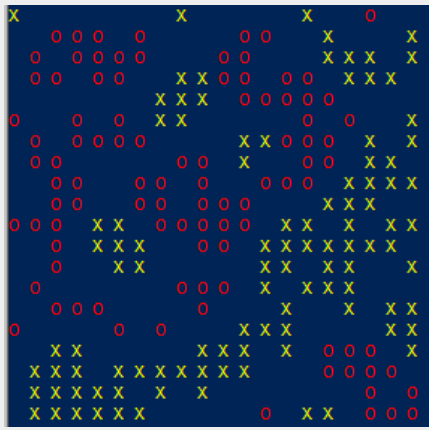
```

Correttezza

La correttezza dell'algorithm può essere provata utilizzando una matrice costante ed andando a controllare che l'algorithm applicato alla matrice da un numero variabile di processi restituisca il medesimo risultato:



Le matrici risultanti sono le seguenti:

Numero processi: 1	Numero processi: 2	Numero processi: 3
Prima iterata	Prima iterata	Prima iterata
		
Seconda iterata	Seconda iterata	Seconda iterata
		

Benchmarking

Il Benchmarking del progetto è stato effettuato su un cluster formato da 4 macchine **m4.large** istanziate su AWS, così da avere 8 vCPUs disponibili. I benchmark sono stati effettuati per valutare la strong scalability e la weak scalability, i test sono i seguenti:

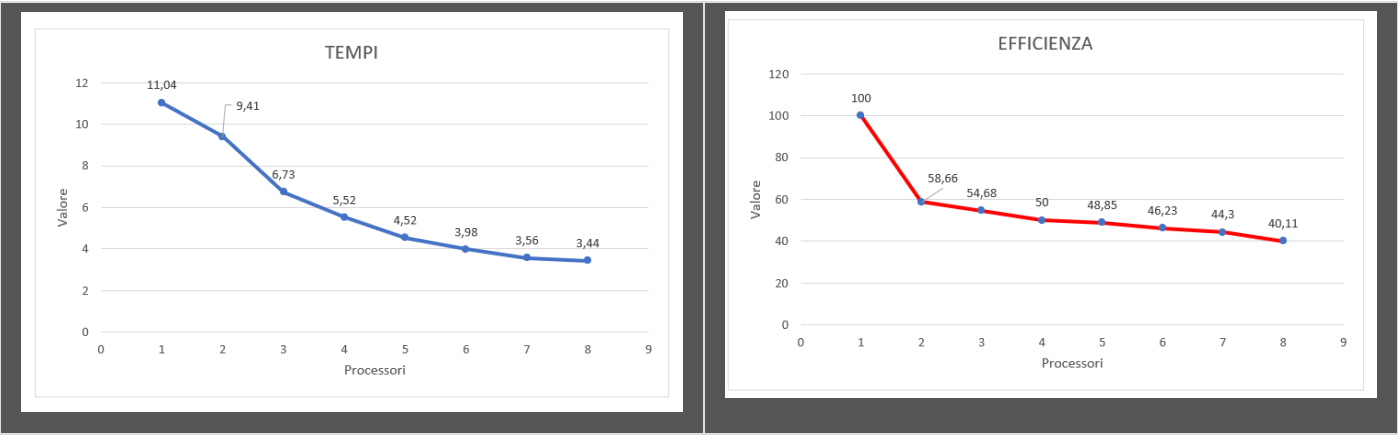
Nel caso della strong scalability i test effettuati sono stati su:

- Matrice 500x500 con numero di processori:1,2,3,4,5,6,7,8 e con otto iterazioni ciascuno
- Matrice 750x750 con numero di processori:1,2,3,4,5,6,7,8 e con otto iterazioni ciascuno
- Matrice 1000x1000 con numero di processori:1,2,3,4,5,6,7,8 e con otto iterazioni ciascuno

La formula utilizzata per calcolare l'efficienza della strong scalability è la seguente: $t1 / (N * tN) * 100\%$, con **t1** che indica il tempo impiegato da 1 processo per effettuare il lavoro, **N** indica il numero di processi e **tN** il tempo impiegato da N processi per effettuare il lavoro.

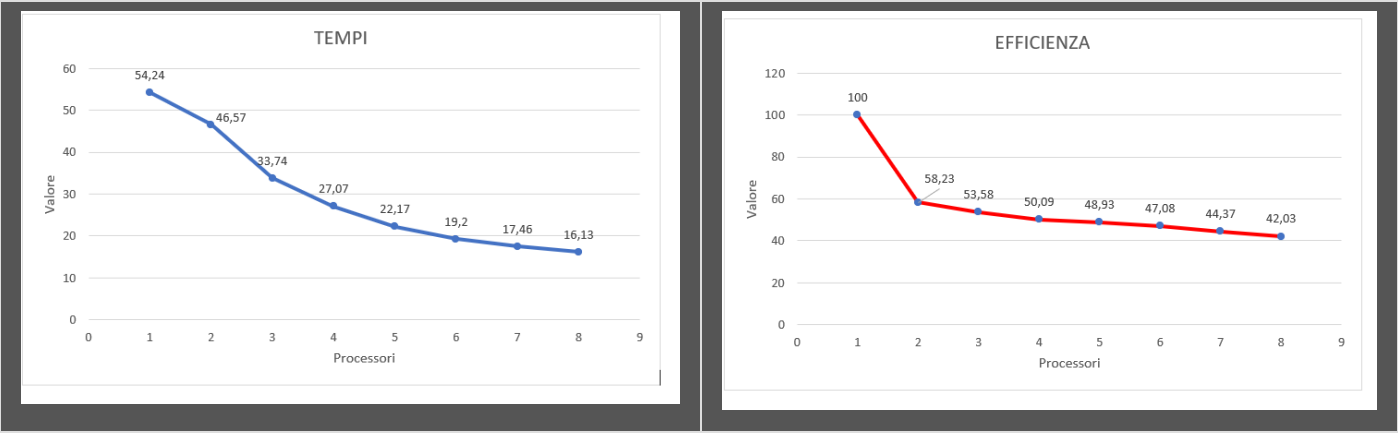
Scalabilità forte su matrice 500x500

vCPUs	1	2	3	4	5	6	7	8
Tempo	11,04	9,41	6,73	5,52	4,52	3,98	3,56	3,44
Efficienza	100,00%	58,66%	54,68%	50%	48,85%	46,23%	44,30%	40,11%



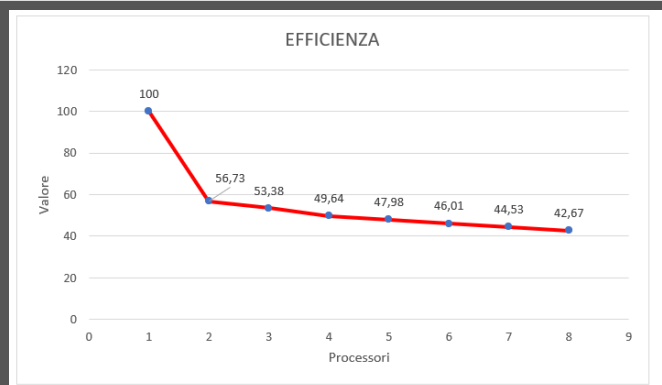
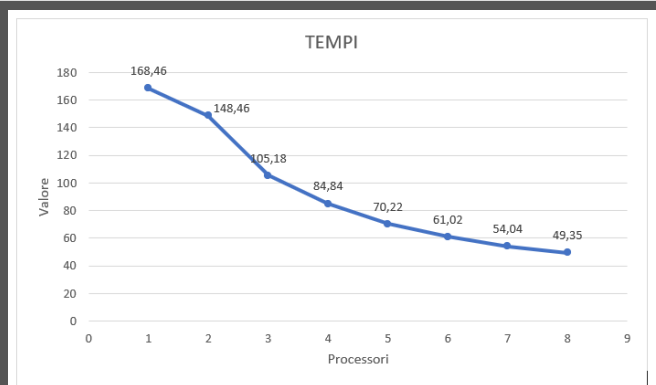
Scalabilità forte su matrice 750x750

vCPUs	1	2	3	4	5	6	7	8
Tempo	54,24	46,57	33,74	27,07	22,17	19,2	17,46	16,13
Efficienza	100,00%	58,23%	53,58%	50,09%	48,93%	47,08%	44,37%	42,03%



Scalabilità forte su matrice 1000x1000

vCPUs	1	2	3	4	5	6	7	8
Tempo	168,46	148,46	105,18	84,84	70,22	61,02	54,04	49,35
Efficienza	100,00%	56,73%	53,38%	49,64%	47,98%	46,01%	44,53%	42,67%

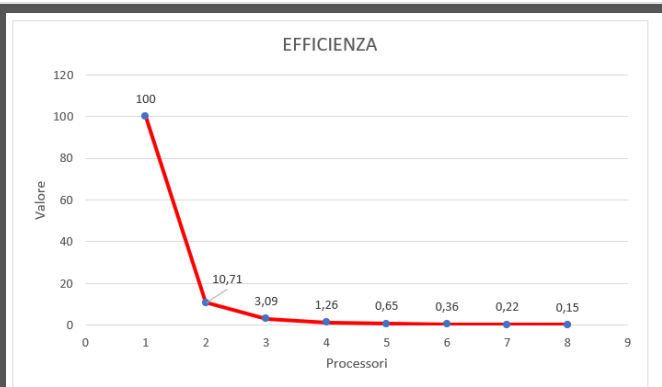
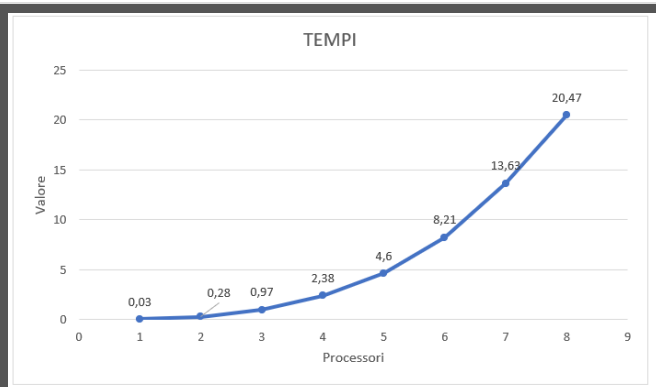


Scalabilità debole

Nel caso della scalabilità debole i test sono stati effettuati iniziando da una matrice composta da 100 righe assegnata ad 1 processo, ed andando poi ad aggiungere 100 righe per ogni processo in più, fino ad arrivare ad una matrice composta da 800 righe assegnata a 8 processi.

La formula utilizzata per calcolare l'efficienza della weak scalability è la seguente: $(t_1 / t_N) * 100\%$, con **t1** che indica il tempo impiegato da 1 processo per effettuare il lavoro e **tN** il tempo impiegato da N processi per effettuare il lavoro.

vCPUs	1	2	3	4	5	6	7	8
Tempo	0,03	0,28	0,97	2,38	4,6	8,21	13,63	20,47
Efficienza	100,00%	10,71%	3,09%	1,26%	0,65%	0,36%	0,22%	0,15%



Esecuzione

Per poter avviare l'esecuzione c'è bisogno di effettuare prima la compilazione del file **Shellings_model.c**:

Readme

```
mpicc Shellings_model.c -o Shellings_model.out
```

Per poi eseguire il compilato:

```
mpirun -np X --mca btl_vader_single_copy_mechanism none --allow-run-as-root Shellings_model.out
```