-Problema: Schelling's model of segregation (Senza semplificazione)

-Name: Aniello Petrosino

 # Indice progettuale:

Per arrivare alla soluzione proposta del problema sono stati affrontati i seguenti passi:

1. Validazione e creazione della matrice
2. Scambio delle celle per il calcolo degli insoddisfatti
3. Calcolo degli insoddisfatti
4. Scambio e riallocazione degli insoddisfatti

1. Validazione e creazione della matrice

Il primo problema affrontato è stato quello di validare inseriti dall'utente, come ad esempio la dimensione della matrice.
Da questo punto di vista il primo controllo effettuato si basa sulla divisione della matrice per il numero di processori in modo tale che da evitare che qualche processore non avesse righe a disposizione.
Il secondo controllo effettuato invece è stato quello relativo agli agenti e alla quantità di spazio libero da lasciare nella matrice, in modo da eseguire con corretta gli spostamenti tra sottomatrici.
All'inizio la quantità di celle libere era calcolata come segue

```
int spazio_disp = dim + dim / 2;
```

In un secondo momento si è passati ad effettuare questo calcolo con l'utilizzo delle percentuali che ha permesso non solo di eseguire molti più controlli in maniera dettagliata ma ha permesso anche di essere molto più precisi nell'individuare quante celle libere lasciare.

```
int support, spazio_disp = (dim * dim) - ((dim * dim) * 30 / 100);
```

Dopo svariati test si è trovato che nella maggior parte dei casi col 30% di celle libere il programma riesce sempre a convergere in una soluzione ottimale.

L'ultimo passaggio è stato quello di dividere i dati tra tutti i processori, per effettuare questo tipo di operazione è stata utilizzata una **struttura** che contiene tutti i dati essenziali di una sottomatrice

```c
typedef struct
{
    int my_dim_x;
    int my_dim_y;
    int my_age_x;
    int my_age_y;
    int sod;
} info_mat;
```

Per utilizzarla è stato necessario **committarla**, come si può vedere nel seguente snippet di codice

```c
MPI_Datatype info_mat_type;
MPI_Datatype info_mat_types[5] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT};
MPI_Aint info_offsets[5];
int info_blocklengths[5] = {1, 1, 1, 1, 1};
info_offsets[0] = offsetof(info_mat, my_dim_x);
info_offsets[1] = offsetof(info_mat, my_dim_y);
info_offsets[2] = offsetof(info_mat, my_age_x);
info_offsets[3] = offsetof(info_mat, my_age_y);
info_offsets[4] = offsetof(info_mat, sod);
MPI_Type_create_struct(5, info_blocklengths, info_offsets, info_mat_types, &info_mat_type);
MPI_Type_commit(&info_mat_type);
```

questa operazione è stata effettuata una sola volta all'interno del programma essendo che la dimensione di quest'ultima non cambierà per tutto il corso dell'esecuzione. Infine, per assegnare e comunicare i dati ai singoli processori, da parte del **Master**, è stato utilizzato un vettore di tipo **info_mat_type** che conterrà in ogni cella i dati relativi ad ogni singolo processore,

```c
//assegna ad ogni processore le corrispettive informazioni
info_x_proc = (info_mat *)malloc(common_size * sizeof(info_mat));
```

che poi successivamente è stato inviato ad ognuno di loro attraverso una

```c
MPI_Scatter(info_x_proc, 1, info_mat_type, &info_generali, 1, info_mat_type, 0, MPI_COMM_WORLD);
```

Alla fine di queste operazioni ogni processore sarà in grado di generarsi la propria matrice e di stamparla per vedere la situazione di partenza.

2 Scambio delle celle per il calcolo degli insoddisfatti

Lo scopo dell'algoritmo è quello di controllare se ogni cella è soddisfatta. Analizzando le casistiche possibili ci si rende conto che in alcuni casi siamo costretti a chiedere ai nostri vicini (cioè myrank−1 e/o myrank+1) di dirci chi sono. Per fare queste richieste sono state usate delle **MPI_Send** e **MPI_Recv** non bloccati

```c
if (myrank != 0)
    MPI_Send(&mymat[0][0], 1, rowtype, myrank - 1, 21, MPI_COMM_WORLD);
if (myrank != common_size - 1)
    MPI_Send(&mymat[info_generali.my_dim_x - 1][0], 1, rowtype, myrank + 1, 20, MPI_COMM_WORLD);
if (myrank != common_size - 1)
{
    ultimariga = (int *)malloc(info_generali.my_dim_y * sizeof(int));
    MPI_Recv(ultimariga, info_generali.my_dim_y, MPI_INT, myrank + 1, 21, MPI_COMM_WORLD, &status);
}
if (myrank != 0)
{
    primariga = (int *)malloc(info_generali.my_dim_y * sizeof(int));
    MPI_Recv(primariga, info_generali.my_dim_y, MPI_INT, myrank - 1, 20, MPI_COMM_WORLD, &status);
}
```

perché nel caso in cui non riceviamo queste informazioni non siamo in grado di proseguire. Altra cosa da tenere in considerazione è la necessità di non farci mandare una singola cella per volta non solo per questioni di tempistiche ma anche di complessità gestionale, e così per ovviare a questo problema abbiamo dichiarato un array continuo che ha permesso di inviare una riga intera

```c
MPI_Datatype rowtype;
MPI_Type_contiguous(info_generali.my_dim_y, MPI_INT, &rowtype);
MPI_Type_commit(&rowtype);

if (myrank != 0)
    MPI_Send(&mymat[0][0], 1, rowtype, myrank - 1, 21, MPI_COMM_WORLD);
if (myrank != common_size - 1)
    MPI_Send(&mymat[info_generali.my_dim_x - 1][0], 1, rowtype, myrank + 1, 20, MPI_COMM_WORLD);
if (myrank != common_size - 1)…
if (myrank != 0)…

MPI_Type_free(&rowtype); //de-allocazione tipo nuovo;
```

3 Calcolo degli insoddisfatti

Arrivati a questo punto si hanno a disposizione tutti i dati che occorrono per il calcolo degli insoddisfatti.
La prima problematica affrontata è stata come memorizzare la posizione delle celle insoddisfatte; per fare questo è stata creata una **struttura** formata dalle coordinate x e y

```c
typedef struct
{
    int x;
    int y;
} insodisfatti;
```

che poi è stata utilizzata per dichiarare 2 array ognuno relativo ad un agente;

```c
array_ins_x = (insodisfatti *)malloc(info_generali.my_age_x * sizeof(insodisfatti));
array_ins_y = (insodisfatti *)malloc(info_generali.my_age_y * sizeof(insodisfatti));
```

Tutta la logica del calcolo è stata demandata alla funzione

```c
calcolo_insodisfatti(mymat, common_size, &risposta_x.celle_libere, &risposta_x.qnt_ins, &risposta_y.qnt_ins, info_generali, array_ins_x,
array_ins_y, primariga, ultimariga, myrank);
```

la quale tiene conto della posizione delle celle che stiamo andando ad analizzare in modo tale da controllare i giusti vicini e dare le giuste percentuali di soddisfazioni, infatti i casi trovati sono 8

```
else if (i == 0 && j == 0) // angolo in alto a sinistra…
else if (i == 0 && j == info_generali.my_dim_y - 1) // angolo in alto a destra…
else if (i == info_generali.my_dim_x - 1 && j == 0) // angolo in basso a sinistra…
else if (i == info_generali.my_dim_x - 1 && j == info_generali.my_dim_y - 1) // angolo in basso a destra…
else if (i != 0 && i != info_generali.my_dim_x - 1 && j == 0) //margine sinistro…
else if (i != 0 && i != info_generali.my_dim_x - 1 && j == common_size - 1) //margine destro…
else if (i == info_generali.my_dim_x - 1) //margine basso…
else if (i == 0) // margine alto…
else // centro…
```

tutte le casistiche presentate precedentemente hanno una struttura uguale alla seguente

```
if (myrank == 0)
{
    if (mymat[i][j + 1] != !mymat[i][j])
        support_sod = support_sod + 33.3;
    if (mymat[i + 1][j] != !mymat[i][j])
        support_sod = support_sod + 33.3;
    if (mymat[i + 1][j + 1] != !mymat[i][j])
        support_sod = support_sod + 33.3;
}
```

questo controllo si riferisce all'angolo in alto a sinistra del master e di conseguenza le celle da controllare adiacenti sono soltanto tre. Il controllo per verificare che il mio vicino mi soddisfi si basa sul verificare che questo sia diverso dal mio negato. Per implementare tutto ciò gli agenti x e y sono stati rappresentati dai valori 1 e 0 e quindi se la condizione spiegata precedentemente risulta essere vera significa che la mia cella adiacente mi possa soddisfare.
Alla fine, dopo aver controllato tutti i vicini vado a vedere se la soddisfazione raggiunta basti affinché quell'agente sia soddisfatto altrimenti lo aggiungo all'array degli insoddisfatti

```
if (support_sod < info_generali.sod)
    aggiungi_insodisfatto(mymat[i][j], array_ins_x, array_ins_y, qnt_ins_x, qnt_ins_y, i, j);
```

questa funzione non fa altro che controllare se l'agente sia 1 o 0 e assegnare la posizione al relativo array

```
void aggiungi_insodisfatto(int agente, insodisfatti *array_ins_x, insodisfatti *array_ins_y, int *qnt_ins_x, int *qnt_ins_y, int i, int j)
{
    if (agente == 1)
    {
        array_ins_x[*qnt_ins_x].x = i;
        array_ins_x[*qnt_ins_x].y = j;
        *qnt_ins_x += 1;
    }
    else
    {
        array_ins_y[*qnt_ins_y].x = i;
        array_ins_y[*qnt_ins_y].y = j;
        *qnt_ins_y += 1;
    }
}
```

4. Scambio e riallocazione degli insoddisfatti

Una volta aver controllato tutte le celle, la fase successiva e finale è stata quella di effettuare lo scambio. Innanzitutto, questa fase è stata gestita dal master, il quale riceve dagli slave il numero di insoddisfatti e li salva all'interno di un array

```
MPI_Gather(&risposta_x.qnt_ins, 1, MPI_INT, global_soddisfazione_x, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Gather(&risposta_y.qnt_ins, 1, MPI_INT, global_soddisfazione_y, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Il numero di insoddisfatti è salvato all'interno di una variabile formata dal seguente tipo

```
typedef struct
{
    int qnt_ins;
    int celle_libere;
} richiesta;
```

che ha permesso anche di tenere traccia di quante celle libere un processore aveva a disposizione e quante insoddisfatti aveva, tutti questi valori sono raccolti nella fase del calcolo insoddisfatti.
Grazie ai precedenti valori il master è in grado di capire se c'è ancora qualche insoddisfatto e in tal caso far effettuare un altro ciclo altrimenti fa terminare il programma.
Nel caso in cui si abbiano ancora degli insoddisfatti la fase successiva è quella di effettuare lo scambio tra processori.
Lo scambio è gestito da una funzione che lavora prima per l'agente 1 e poi con l'agente 0.

```
void change_for_process(int myrank, int common_size, int **mymat, richiesta *array, insodisfatti *array_ins, int agente, info_mat *info_generali, richiesta *rispos
{
    int qnt_ins = risposta->qnt_ins;
    int celle_vuote = risposta->celle_libere;
    /**
    Il compito di effettuare gli scambi è demanato al master il quale randomicamente scegle su quale processore spostare gli insoddisfati.
    Al processore scelto gli verranno ascegnati tutti gli agenti che può prendere in base alle celle libere, mentre gli agenti che non verranno
    soddisfatti o dovranno aspettare il ciclo successivo o probabilmente dopo questo scambio saranno soddisfatti
    **/
    if (myrank == 0)
        for (int i = 0; i < common_size; i++)
        {
            int procs, supp_pos = 0;
            if (array[i].qnt_ins > 0)
            {
                procs = rand() % common_size;
                if (array[procs].celle_libere > 0)
                {
                    while (array[procs].celle_libere > 0 && array[i].qnt_ins > 0)
                    {
                        array[procs].celle_libere -= 1;
                        array[i].qnt_ins -= 1;
                    }
                }
            }
        }

    MPI_Scatter(array, 1, type, risposta, 1, type, 0, MPI_COMM_WORLD);
    /**Dopo gli scambi vengono comunicati ai processori che procedono nel libere e assegnare le celle**/
    for (int i = qnt_ins; i > risposta->qnt_ins; i--)
    {
        mymat[array_ins[i - 1].x][array_ins[i - 1].y] = 1000;
        if (agente == 1)
            info_generali->my_age_x -= 1;
        else
            info_generali->my_age_y -= 1;
    }

    for (int i = celle_vuote; i > risposta->celle_libere; i--)
    {
        assegna(mymat, agente, info_generali->my_dim_x, info_generali->my_dim_y);
        if (agente == 1)
            info_generali->my_age_x += 1;
        else
            info_generali->my_age_y += 1;
    }
}
```

Il funzionamento di questa funzione è molto semplice, lo scambio è gestito dal master il quale randomicamente sceglie il processore ricevente degli insoddisfatti di qualcun

altro, il ricevente prenderà in input il numero massimo di agenti, quelli restanti insoddisfatti dovranno o aspettare il ciclo successivo per cambiare sottomatrice oppure dopo questa fase di scambio potranno essere automaticamente soddisfatti.
Dopo di che ogni agente libererà le celle relative agli agenti che "se ne sono andati" e assegnerà le celle ai nuovi agenti.

Il programma ciclerà fin quando tutti gli agenti non sono soddisfatti.

 # Esecuzione e parametri

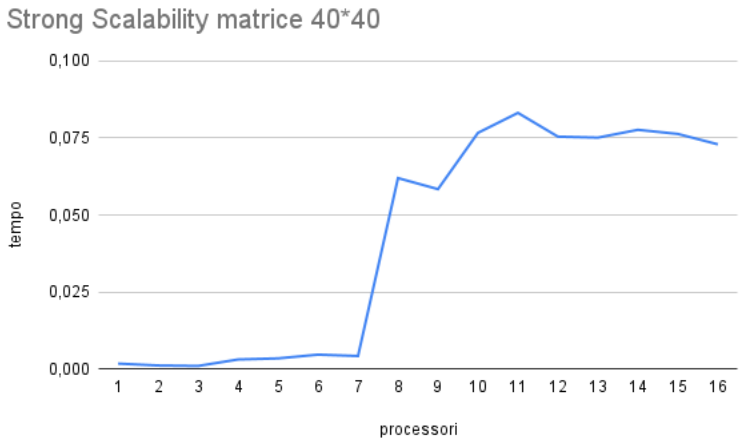
Per eseguire il programma occorre prima creare l'eseguibile eseguendo il seguente comando

```
mpicc Schelling-s-model-of-segregatiom.c -o Schelling-s-model-of-segregatiom
```

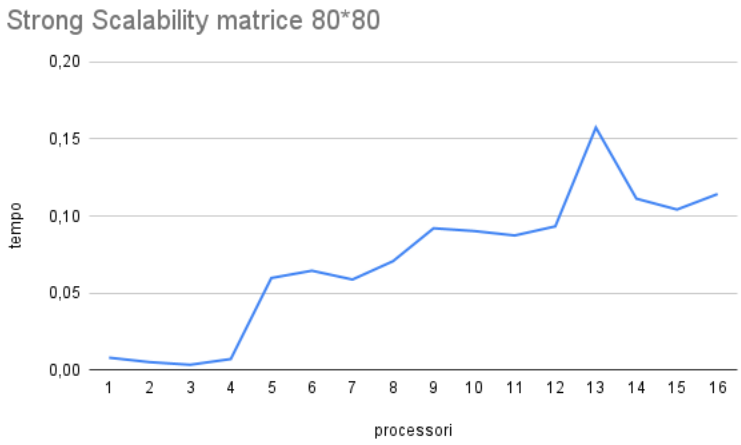
Per poi lanciarlo con il seguente comando e scrivere in input i dati richiesti dal programma

```
mpirun -np 2 ./Schelling-s-model-of-segregatiom
```

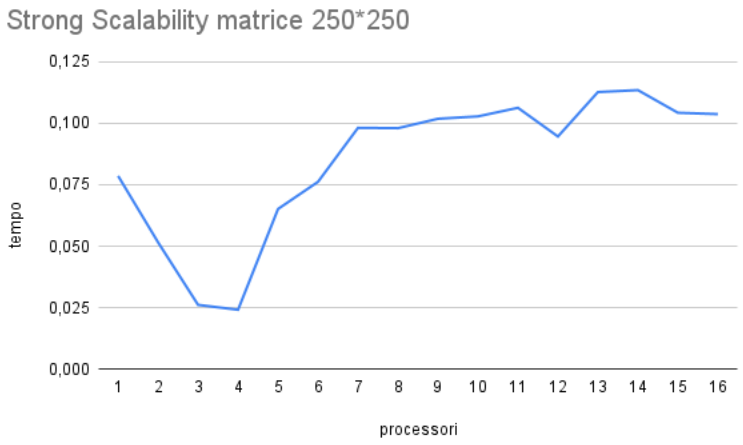
Per i test di ****scalabilità forte**** sono stati scelti i seguenti input:
Matrice 40*40, agenti x=580 y=540 sod=60



Matrice 80*80, agenti x=2280 y=2200 sod=70

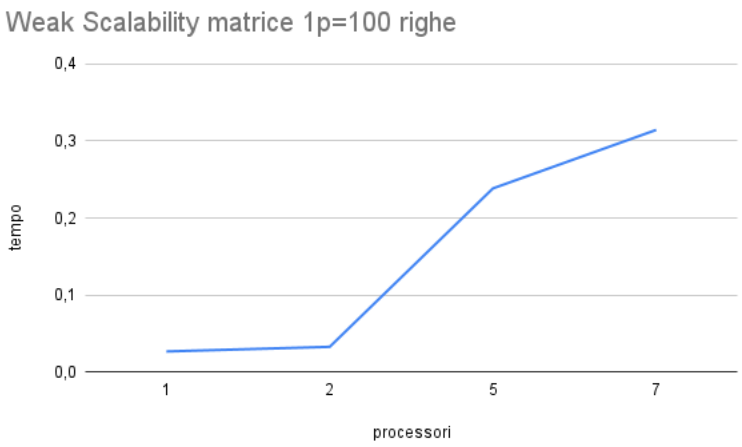


Matrice250*250, agenti x=21950 y=21800 sod=70

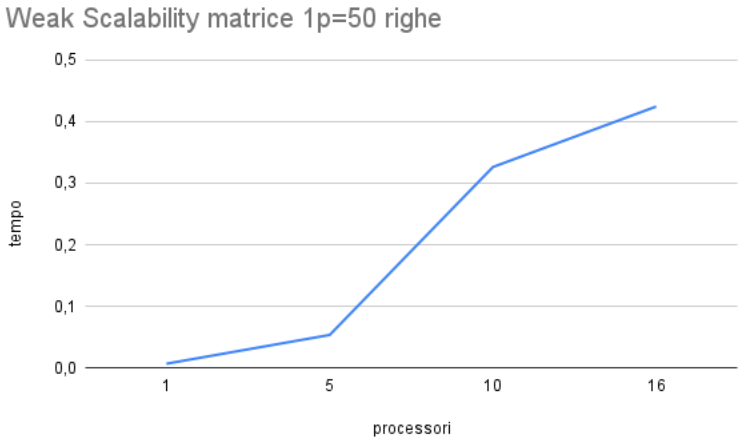


Per i test di **scalabilità debole** sono stati fatti due test,

-il primo assegnando ad per ogni processore 50 righe



-il secondo assegnando ad per ogni processore 100 righe



Conclusione

In conclusione, per i test eseguiti si può notare che la parallelizzazione ha i suoi pro e contro questo in base al tipo di problema e la mole di dati passati.
Per quello che riguarda il mio caso di studi, con la quantità di dati passati e l'algoritmo implementato la parallelizzazione non presenta una soluzione ottimale da un punto di vista prestazionale questo perché con l'aumentare dei processori ci troviamo ad aumentare la probabilità degli scambi e quindi di conseguenza aumenta la possibilità che a ogni iterazione una cella che era soddisfatta diventi insoddisfatta.