Schelling-s-Model-of-Segregation-PCPC

Progetto per l'esame di *Programmazione Concorrente, Parallela e sul Cloud* dell'anno di corso *2020/2021*.

Laurea magistrale in Computer Science, curriculum in Cloud Computing.

Lorenzo Petrazzuolo 0522500894

23/08/2021

Indice

1 Introduzione

1.1 Il modello di segregazione di Schelling

Nel 1971, l'economista americano Thomas Schelling creó un modello basato su agenti evidenziando il fatto che la segregazione era anche risultato di comportamenti involontari. Piú precisamente, il modello poneva la luce sulla possibilitá che un individuo o agente potesse allontanarsi involontariamente dagli altri agenti nel tempo per motivi razziali, economici o sociali.

Sebbene il modello sia abbastanza semplice, fornisce uno sguardo affascinante su come gli individui potrebbero auto-segregarsi, anche quando non hanno un desiderio esplicito di farlo.

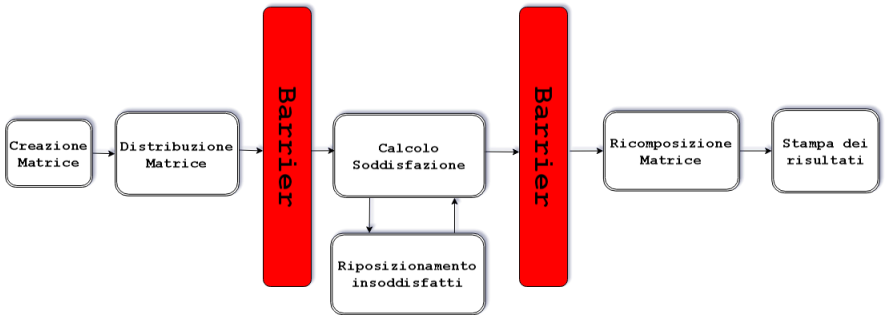
1.2 Descrizione del problema

Nel seguente progetto si propone un'implementazione di una simulazione del modello di Shelling. Una volta impostati una serie di parametri, il programma dovrá essere in grado di descrivere graficamente le caratteristiche del modello. Piú precisamente si suppone di avere due tipi di agenti ***X*** e ***O*** che andranno a comporre la popolazione di una matrice **NxM**. Fissata una soglia di soddisfazione, ogni agente si dirá soddisfatto se la percentuale delle celle limitrofe é del suo stesso tipo, altrimenti sará un agente insoddisfatto e dovrá essere spostato randomicamente in un'altra cella vuota della matrice. Questo avviene ad ogni round fino a che non si avvera una delle seguenti condizioni:

- Tutti gli agenti della matrice sono soddisfatti
- Il numero di round ha oltrepassato il limite massimo di round

2 Dettagli implementativi

Per la risoluzione del problema é stato scelto di usare **C** come linguaggio di programmazione e la libreria **MPI (Message Passing Interface)** per lavorare in logica distribuita. Dal seguente diagramma di flusso é possibile evidenziare i passi principali di computazione che hanno permesso di trovare una soluzione al quesito posto. Successivamente verranno illustrati i vari punti nel dettaglio.



2.1 Creazione Matrice

Per effettuare la generazione della matrice é importante andare a definire il numero preciso di ***X***, ***O*** e ' ' (celle vuote). Questo viene calcolato tramite il prodotto tra righe, colonne e valore percentuale di presenza di agenti di un certo tipo.

```
int E = (ROWS*COLUMNS*PERC_E);
int O = (ROWS*COLUMNS-E)*PERC_O, X = (ROWS*COLUMNS-E)*PERC_X;
if(ROWS*COLUMNS != (O+E+X)) E+=(ROWS*COLUMNS)-(O+E+X);
```

Mediante la funzione *create_matrix* é possibile generare una matrice di dimensioni fissate **ROWSxCOLUMS** dove i valori delle celle sono scelti randomicamente tramite la funzione **rand()** il cui seed viene fissato precedentemente in funzione del numero di secondi dell'ora locale oppure tramite un valore fissato scelto dall'utente. Ogniqualvolta viene inserito un agente, viene decrementato il suo contatore fino a riempire la matrice con le diverse categorie di agenti.

```
void create_matrix(char* mat, int E, int O, int X){

    int i, val_rand=0;
    //Seed per funzione di rand()
    time_t now = time(NULL);
    struct tm *tm_struct = localtime(&now);
    srand ( localtime(&now)->tm_sec );

    for(i=0; i<ROWS*COLUMNS; i++){
        val_rand=rand()%3;
```

```
        if(val_rand==0 && O>0){
            mat[i] = 'O';
            O--;
        }else if(val_rand==1 && X>0){
            mat[i] = 'X';
            X--;
        }else if(val_rand==2 && E>0){
            mat[i] = '␣';
            E--;
        }else{

            if(E>0){
                mat[i] = '␣';
                E--;
            }else if((val_rand==0 || val_rand==2)
                        && (E==0 || O==0)){
                mat[i] = 'X';
                X--;
            }else if((val_rand==1 || val_rand==2)
                        && (E==0 || X==0)){
                mat[i] = 'O';
                O--;
            }
        }
    }
}
```

2.2 Distribuzione Matrice

Affinché si distribuisca la matrice in maniera intelligente é importante andare a definire una struttura dati dedicata che permetta ad ogni processo di salvare la propria sottomatrice e altri indici della stessa utili per il calcolo della soddisfazione degli agenti.

```
typedef struct{
    //Sottomatrice assegnata ad ogni thread
    char *submatrix;

    //Valori per la suddivisione e ricomposizione matrice
    //in funzione di una Scatter
    int *scounts_scatter;
    int *displ_scatter;

    //Valori per la suddivisione e ricomposizione matrice
    //in funzione di una Gather
    int *scounts_gather;
```

```
    int *displ_gather;

} Info_submatrix;
```

La distribuzione della matrice di partenza avviene in base al numero di processi coinvolti e non é altro che la sua suddivisione in righe. Al fine di calcolare in un secondo momento la soddisfazione degli agenti, si é scelto di ampliare la sezione di righe dedicata ai singoli thread di una o due righe supplementari, rispettivamente una per il primo (MASTER) e l'ultimo processo, due per i processi a cui sono dedicate sezioni "centrali" della matrice. In merito a ció, **scount_scatter** e **displs_scatter** rappresentano rispettivamente il numero di elementi e gli indici delle sottomatrici con righe supplementari di ogni thread, **scount_gather** e **displs_gather** invece rappresentano il numero di elementi e gli indici delle sottomatrici effettive quindi senza righe supplementari dedicate ai singoli processi. I primi due array di valori utili per il calcolo della soddisfazione degli agenti, gli altri due utili per la ricomposizione della matrice.

```
//Distribuzione matrice in sottomatrici per thread: calcolo
//size e displacements di tutte le sottomatrici dei vari thread
void distribute_matrix(Info_submatrix t_mat, int numproc){

    if(numproc>1){
        int i=0;
        int size_r_int = ROWS / numproc;
        int size_r_rest = ROWS % numproc;
        for(i=0;i<numproc;i++){
            if(i<size_r_rest){
                t_mat.scounts_scatter[i]= (size_r_int+1) * COLUMNS;
                t_mat.scounts_gather[i] = (size_r_int+1) * COLUMNS;
            }
            else{
                t_mat.scounts_scatter[i] = size_r_int * COLUMNS;
                t_mat.scounts_gather[i] = size_r_int * COLUMNS;
            }
            if(i==0){
                t_mat.scounts_scatter[i] += COLUMNS;
                t_mat.displ_scatter[i] = 0;
            }else if(i==(numproc-1)){
                t_mat.scounts_scatter[i] += COLUMNS;
                t_mat.displ_scatter[i] = ROWS*COLUMNS -
                                         t_mat.scounts_scatter[i];
            }else{
                t_mat.scounts_scatter[i] += COLUMNS*2;
                t_mat.displ_scatter[i] = t_mat.displ_scatter[i-1] +
                                         t_mat.scounts_scatter[i-1] -
                                         COLUMNS*2;
                }
```

```
            t_mat.displ_gather[i] = i==0 ? 0 :
                        t_mat.displ_gather[i-1] +
                        t_mat.scounts_gather[i-1];
        }
    }else{ //Un solo processo
        t_mat.scounts_scatter[0] = ROWS * COLUMNS;
        t_mat.displ_scatter[0] = 0;
        t_mat.scounts_gather[0] = ROWS*COLUMNS;
        t_mat.displ_gather[0] = 0;
    }
}
```

A questo punto, in base ai valori precedentemente calcolati, é possibile suddividere la matrice in sottomatrici e inviare ogni sezione al processo dedicato tramite una **Scatterv**.

```
//Suddivisione effettiva della matrice per i vari thread
MPI_Scatterv(mat, t_mat.scounts_scatter, t_mat.displ_scatter,
MPI_CHAR, t_mat.submatrix, t_mat.scounts_scatter[myrank],
MPI_CHAR, 0, MPI_COMM_WORLD);
```

2.3 Calcolo soddisfazione

Il calcolo della soddisfazione degli agenti viene effettuato da *satisfaction_step* che restituisce sia il numero di celle soddisfatte sia una struttura dati dedicata alla memorizzazione di celle insoddisfatte e celle vuote, di seguito presentata.

```
typedef struct{
    //Array composto dagli indici delle celle insoddisfatte
    char *unsatisfied;
    //Numero di celle insoddisfatte
    int n_unsatisfied;
    //Array composto dagli indici delle celle vuote
    int *freeslots;
    //Numero di celle vuote
    int n_freeslots;

} Info_cellpositions;
```

Per calcolare la soddisfazione di un agente é importante verificare dapprima il rango del processo per capire se si tratta di un thread a cui é stata dedicata una sottomatrice "centrale" oppure no. Una volta verificato ció, se la cella é piena allora, in base all'indice, verrá chiamata un'apposita funzione che permetterá di calcolare i valori delle celle vicine:

1. per una cella angolo
    - **calc_firstangle**

- **calc_secondangle**
- **calc_thirdangle**
- **calc_fourthangle**

2. per una cella sulla cornice
   - **calc_Ledge** = Cornice sinistra
   - **calc_Redge** = Cornice destra
   - **calc_Aedge** = Cornice in alto
   - **calc_Bedge** = Cornice in basso
3. per una cella centrale
   - **calc_center**

Ad ognuna di queste funzioni verrá passata sia la propria sottomatrice che l'indice della cella in esame. Dopo aver calcolato il numero di celle vicine aventi come agente quello della cella passata, il calcolo della soddisfazione sará dato dal confronto di due variabili:

```
return (similar_cells>=similarity);
```

dove similar_cells é il numero di celle vicine con lo stesso tipo di agente di quello in esame, mentre similarity[1] rappresenta il grado di soddisfazione dato da:

```
float similarity = PERC_SIM*N; //Grado di soddisfazione
```

Se il numero di celle vicine é maggiore o uguale al valore di similarity allora la cella in esame é soddisfatta e verrá restituito 1 come valore.

```
if(i==0) {
    if(calc_firstangle(t_mat)) local_satisfied++;
    else{
     cellpos.unsatisfied[cellpos.n_unsatisfied]=t_mat.submatrix[i];
     cellpos.freeslots[cellpos.n_freeslots]=i;
     cellpos.n_freeslots++;
     cellpos.n_unsatisfied++;
     t_mat.submatrix[i]='␣';
    }
}
```

Prendendo come esempio il calcolo di **first_angle**, se il valore restituito é 1 allora viene incrementata la variabile di soddisfatti locali altrimenti:

1. L'indice della cella insoddisfatta viene salvato nell'array delle celle insoddisfatte;

[1]Note: Il valore PERC_SIM é la percentuale di soddisfazione che viene fissata prima della fase di compilazione (Si veda la sezione 'Note sull'implementazione' 3). Il valore N varia a seconda della posizione della cella: 3 per cella angolo, 5 per cella cornice, 8 per cella centrale

2. L'indice della cella insoddisfatta viene salvato nell'array di celle vuote;

3. Il numero di celle vuote viene incrementato;

4. Il numero di celle insoddisfatte viene incrementato;

5. La cella insoddisfatta viene sostituita con una cella vuota.

Se invece la cella é vuota:

1. L'indice della cella vuota viene salvato nell'array di celle vuote;

2. Il numero di celle vuote viene incrementato.

Nel momento in cui ogni thread ha calcolato il numero delle celle soddisfatte della proprio sottomatrice, tramite una **MPI_Allreduce** ogni processo determinerá il valore della soddisfazione globale, cioé quella dell'intera matrice

```
MPI_Allreduce(&local_satisfied, &global_satisfied, 1, MPI_INT,
MPI_SUM, MPI_COMM_WORLD);
```

2.4 Riposizionamento insoddisfatti

Il riposizionamento degli insoddisfatti viene effettuato dalla funzione *displacements_step*. L'obiettivo qui é quello di aggiornare tutti i processi circa l'insieme delle celle insoddisfatte e l'insieme di celle vuote di tutti gli altri thread. Questo permette di tenere traccia sia del numero e degli indici delle celle vuote sia del numero e degli indici delle celle insoddisfatte cosí da costruire un array di valori che, una volta randomizzato, dia la possibilitá di ricollocare gli agenti insoddisfatti in posizioni randomiche dell'intera matrice.

Piú precisamente verranno fatte eseguire due **Allgather** rispettivamente per il calcolo del numero di celle insoddisfatte per ogni thread e il calcolo del numero di celle vuote per ogni thread.

```
MPI_Allgather(&cellpos.n_unsatisfied, 1, MPI_INT,
number_of_unsatisfied_per_th, 1, MPI_INT, MPI_COMM_WORLD);

MPI_Allgather(&cellpos.n_freeslots, 1, MPI_INT,
number_of_freeslots_per_th, 1, MPI_INT, MPI_COMM_WORLD);
```

Grazie alle informazioni contenute in **number_of_unsatisfied_per_th** e **number_of_freeslots_per_th** é possibile calcolare i displacements delle celle insoddisfatte e quelli delle celle vuote insieme ai quali si vanno a definire due array

- **all_freeslots** = Array di indici di celle vuote dell'intera matrice
- **total_unsat_freeslots** = Array di indici di celle insoddisfatte dell'intera matrice

tramite due **Allgatherv**

```
MPI_Allgatherv(cellpos.freeslots, cellpos.n_freeslots, MPI_INT,
all_freeslots, number_of_freeslots_per_th, displs_freeslots,
MPI_INT, MPI_COMM_WORLD);

MPI_Allgatherv(cellpos.unsatisfied, cellpos.n_unsatisfied,
MPI_CHAR, total_unsat_freeslots, number_of_unsatisfied_per_th,
displs_unsatisfied, MPI_CHAR, MPI_COMM_WORLD);
```

Siccome il numero di celle vuote é maggiore o uguale al numero di celle insoddisfatte é importante riempire **total_unsat_freeslots** (che al momento contiene solo valori di agenti insoddisfatti) di celle vuote cosí da poterlo suddividere in sezioni, ognuna dedicata ad un processo.

```
//Prima di randomizzare l'array di celle insoddisfatte,
//viene riempito con ' ' (celle vuote) per far coincidere
//la size dell'array con il numero di celle vuote totali
for(i=unsatisfied; i<sum_number_freeslots; i++)
    total_unsat_freeslots[i]='␣';
```

A questo punto é possibile randomizzare l'array,

```
void randomize ( char *arr, int n )
{
    for (int i = n-1; i > 0; i--)
    {
        int j = rand() % (i+1);
        swap(&arr[i], &arr[j]);
    }
}
```

sostituire i valori della sottomatrice con quelli presenti nell'array randomizzato

```
//Vengono sostituiti i valori nei freeslots delle
//varie sottomatrici con i valori delle sezioni di
//total_unsatisfied in funzione del proprio rank
for(i=0;i<cellpos.n_freeslots;i++)
    t_mat.submatrix[cellpos.freeslots[i]]=
      total_unsat_freeslots[displs_freeslots[myrank]+i];
```

e aggiornare le righe supplementari con i nuovi valori ricollocati nella loro nuova posizione all'interno della matrice.

```
//Prima di assegnare i nuovi valori delle celle insoddisfatti
//alle matrici dei vari thread, vengono aggiornate le righe
//supplementari utili al calcolo della soddisfazione al
//successivo round

```

```
//Rank==0 -> MASTER aggiornare solamente una riga supplementare
//cio\'e l'ultima della sottomatrice
if(myrank==0)
   for(i=0; i<number_of_freeslots_per_th[1];i++){
     if(all_freeslots[displs_freeslots[1]+i]>=(COLUMNS*2)) break;
     else if(all_freeslots[displs_freeslots[1]+i]>=COLUMNS){
          t_mat.submatrix[t_mat.scounts_scatter[myrank]-
           COLUMNS+(all_freeslots[displs_freeslots[1]+i]%COLUMNS)]
           = total_unsat_freeslots[displs_freeslots[1]+i];
     }
   }
//Rank==(totale processi - 1) -> ULTIMO THREAD quindi
//bisogna aggiornare solamente una riga cio\'e la prima
//della sottomatrice
else if(myrank==(numproc-1)){
   for(i=0; i<number_of_freeslots_per_th[numproc-2];i++)
       if(all_freeslots[displs_freeslots[numproc-2]+i]>=
         (t_mat.scounts_scatter[numproc-2]-(COLUMNS*2))){
         t_mat.submatrix[all_freeslots[(displs_freeslots[numproc-2]+i)]%COLUMNS] =
         total_unsat_freeslots[displs_freeslots[numproc-2]+i];
       }
}
//Rank compreso tra 0 e l'ultimo thread quindi bisogna
//aggiornare due righe supplementari, la prima e
//l'ultima della sottomatrice
else{
   //PRIMA RIGA
   for(i=0; i<number_of_freeslots_per_th[myrank-1];i++)
       if(all_freeslots[displs_freeslots[myrank-1]+i]>=
       (t_mat.scounts_scatter[myrank-1]-(COLUMNS*2)) &&
       all_freeslots[displs_freeslots[myrank-1]+i]<
       (t_mat.scounts_scatter[myrank-1]-COLUMNS)){
           t_mat.submatrix[all_freeslots[displs_freeslots[myrank-1]+i]%COLUMNS] =
           total_unsat_freeslots[displs_freeslots[myrank-1]+i];
       }
   //ULTIMA RIGA
   for(i=0; i<number_of_freeslots_per_th[myrank+1];i++)
       if(all_freeslots[displs_freeslots[myrank+1]+i]>=
       COLUMNS && all_freeslots[displs_freeslots[myrank+1]+i]<
       (COLUMNS*2)){
          t_mat.submatrix[t_mat.scounts_scatter[myrank]-COLUMNS+
          (all_freeslots[displs_freeslots[myrank+1]+i]%COLUMNS)]=
          total_unsat_freeslots[displs_freeslots[myrank+1]+i];
       }
}
```

2.5 Ricomposizione Matrice

Se tutti gli agenti sono soddisfatti o é stato superato il numero di round massimo, allora si esce dal ciclo, viene ricomposta la matrice finale e vengono stampati i risultati.

La ricomposizione della matrice avviene tramite la funzione *recompose_mat* che non fa altro che generare un array con i valori della sottomatrice assegnata al processo e inviarlo tramite una **Gatherv** al master.

```
MPI_Gatherv(submatGather, t_mat.scounts_gather[myrank], MPI_CHAR,
final_mat, t_mat.scounts_gather, t_mat.displ_gather, MPI_CHAR,
0, MPI_COMM_WORLD);
```

2.6 Stampa dei risultati

La stampa dei risultati é l'ultimo passo del flusso di operazioni del programma. I dati stampati a video riguardano:

- Matrice iniziale e matrice finale
- Risultato ottenuto dalla computazione
  - Numero di celle soddisfatte
  - Percentuale di soddisfazione ottenuta
  - Tempo di computazione
  - Numero di round raggiunto
- Parametri in input
  - Taglia matrice
  - Percentuale valori X, O e celle vuote
  - Percentuale di soddisfazione

3 Note sull'implementazione

Per rendere l'implementazione grafica piú accattivante é stato pensato di dare la possibilitá di visualizzare la matrice aggiornata ad ogni round, ovviamente a discapito delle prestazioni. Per questo motivo si consiglia di optare per questa scelta qualora non si vogliano misurare le performance del programma.

Per fare ció é fondamentale la modifica dei valori **DELAY**, **PRINT_ROUNDS** e **PERFORMANCE**. Il primo per sospendere il processo master per secondi dopo la stampa della matrice aggiornata, il secondo valore per consentire la visualizzazione a video della matrice risultante mentre il terzo per autorizzare la stampa a sfavore delle prestazioni.

```
//Delay di attesa per round
#define DELAY 0

//'1' se si vuole stampare la matrice risultante ad ogni round
#define PRINT_ROUNDS 1

//'1' se si vuole calcolare matrice risultante senza stampe
//a favore delle prestazioni
#define PERFORMANCE 0
```

Inoltre, prima di effettuare la compilazione, é importante settare i valori utili per la costruzione della matrice, per il calcolo della soddisfazione e per il numero di round di computazione massimi.

```
//Numero Colonne
#define COLUMNS 30

//Numero Righe
#define ROWS 30

//Numero Round Massimi
#define N_ROUND_MAX 300

//Percentuale di 'O'
#define PERC_O 0.5

//Percentuale di 'X'
#define PERC_X (1-PERC_O)

//Percentuale di celle vuote
#define PERC_E 0.3

//Percentuale di soddisfazione
#define PERC_SIM 0.3
```

3.1 Compilazione

Una volta settati tutti i parametri é possibile effettuare la fase di compilazione del codice con il seguente comando:

```
mpicc -o prog prog.c
```

I requisiti di successo per quanto riguarda la compilazione e di conseguenza anche l'esecuzione del codice sono l'installazione di:

- Ubuntu Linux 18.04 LTS
- OpenMPI

3.2 Esecuzione

Per effettuare la fase di esecuzione del codice, una volta generato il file .out durante la fase di compilazione é possibile eseguirlo con il seguente comando:

```
mpirun --allow-run-as-root -np N prog
```

N rappresenta il numero di processi.

4 Benchmarks

In fase di Benchmarking é stata misurata la scalabilitá del programma in termini delle due nozioni principali di High Performance Computing, cioé *Strong Scalability* e *Weak Scalability*. I test sono stati effettuati su un cluster di quattro macchine con sistema operativo *Ubuntu Linux 18.04 LTS*, della famiglia *t2.xlarge* con *16GB di memoria* e *4 vCPU* ognuna.

4.1 Strong Scalability

La Strong Scalability puó essere definita come il modo in cui il tempo di computazione varia, nella risoluzione di un problema di dimensione fissata, al variare del numero di processori. La misurazione di questo tipo di scalabilitá permette di capire, da un lato, di quanto varia il tempo di computazione di un programma aumentando la parte di codice in parallelo rispetto a quella sequenziale e, dall'altro lato, qual é il limite superiore di processi coinvolti oltre il quale si rischia di estendere i tempi di calcolo a causa dell'overhead parallelo. Per definizione, la misurazione deve essere effettuata in base ad una dimensione fissata del problema calcolato da un numero variabile di processi. Per questo motivo é stato scelto di testare l'efficienza della Strong Scalability in funzione di una formula:

$$\frac{t1}{(N * tN)} * 100\% \tag{1}$$

- t1 = tempo di computazione di un singolo processo
- N = numero processi coinvolti
- tN = tempo di computazione di N processi

4.1.1 Test-1 K/2 (2500x2000)

Righe	Colonne	Threads	Tempo 1	Tempo 2	Tempo 3	Efficienza*
2500	2000	1	12,613839	12,620761	12,610214	100%
2500	2000	2	6,646243	6,644074	6,629937	94,97%
2500	2000	4	4,477803	**4,455112**	4,467496	70,46%
2500	2000	6	6,948272	6,7789	6,839108	30,27%
2500	2000	8	7,328779	7,323284	7,192262	21,52%
2500	2000	10	8,793001	8,318929	8,563753	14,35%
2500	2000	12	8,147662	7,754639	8,234876	12,77%
2500	2000	14	6,92885	7,097258	6,946975	12,70%
2500	2000	16	8,136461	7,871201	8,091794	9,69%

*Per il calcolo dell'efficienza sono stati presi in considerazione i tempi peggiori

4.1.2 Test-2 K (5000x2000)

Righe	Colonne	Threads	Tempo 1	Tempo 2	Tempo 3	Efficienza*
5000	2000	1	22,807374	22,706562	22,817516	100%
5000	2000	2	15,707718	15,737176	15,815975	72,13%
5000	2000	4	9,697376	**9,660594**	9,675218	58,82%
5000	2000	6	14,07761	13,323792	13,267828	27,01%
5000	2000	8	19,111812	18,422374	20,341324	14,02%
5000	2000	10	15,792132	15,734468	15,559272	14,44%
5000	2000	12	18,217392	18,031201	17,311761	10,43%
5000	2000	14	14,850863	14,927343	14,749345	10,91%
5000	2000	16	16,271422	15,993416	15,966075	8,76%

*Per il calcolo dell'efficienza sono stati presi in considerazione i tempi peggiori

4.1.3 Test-3 2K (10000x2000)

Righe	Colonne	Threads	Tempo 1	Tempo 2	Tempo 3	Efficienza*
10000	2000	1	49,179175	49,347264	49,205463	100%
10000	2000	2	35,39052	35,373937	35,464182	69,57%
10000	2000	4	21,200705	**21,17713**	21,258909	58,03%
10000	2000	6	32,048546	30,198325	30,616614	25,66%
10000	2000	8	33,514537	32,541497	33,063533	18,40%
10000	2000	10	37,897307	38,500395	37,473097	12,81%
10000	2000	12	34,740301	34,479201	35,489891	11,58%
10000	2000	14	32,819148	32,225567	32,608265	10,74%
10000	2000	16	32,937963	33,445424	33,151032	9,22%

*Per il calcolo dell'efficienza sono stati presi in considerazione i tempi peggiori

4.2 Weak Scalability

La Weak Scalability puó essere definita come il modo in cui il tempo di computazione varia in funzione del numero di processori, per una dimensione del problema fissata per processore. La misurazione di questo tipo di scalabilitá permette di capire quanto un programma é scalabile in funzione dell'aumento delle risorse richieste per processore. L'obiettivo é quello di avere dei tempi di esecuzione costanti con un aumento lineare della dimensione del problema. La misurazione deve essere effettuata in base ad un aumento fissato della dimensione del problema per il numero di processori coinvolti. Per questo motivo é stato scelto di testare l'efficienza della Weak Scalability in funzione di una formula:

$$\frac{t1}{tN} * 100\% \tag{2}$$

- t1 = tempo di computazione di un singolo processo
- N = numero processi coinvolti
- tN = tempo di computazione di N processi

Righe	Colonne	Threads	Tempo 1	Tempo 2	Tempo 3	Efficienza*
2000	2000	1	8,607591	8,625888	8,645915	100%
4000	2000	2	12,138668	12,213413	12,202741	70,79%
6000	2000	4	12,581904	12,588319	12,614354	68,54%
8000	2000	6	22,022071	21,743886	22,085999	39,14%
10000	2000	8	32,980166	31,744703	32,340681	26,21%
12000	2000	10	40,823197	39,834169	40,741302	21,17%
14000	2000	12	49,732282	51,737642	51,075773	16,71%
16000	2000	14	59,512686	57,37106	58,915781	14.52%
18000	2000	16	69,99322	68,117497	69,723649	12,35%

*Per il calcolo dell'efficienza sono stati presi in considerazione i tempi peggiori

Risultati I risultati mostrati permettono di evidenziare i miglioramenti prestazionali che si possono avere con l'introduzione della programmazione parallela. Piú precisamente, per quanto riguarda la Strong Scalability, é possibile notare nei tre test effettuati che il tempo migliore é stato ottenuto dall'esecuzione del programma da 4 processori sottolineando il fatto che la distribuzione del lavoro é stata svolta nel migliore dei modi. Purtroppo, con l'aumento dei processori coinvolti, c'é stato un notevole incremento della comunicazione parallela che ha portato ad aumentare i tempi di elaborazione. Anche riguardo la Weak Scalability sono stati ottenuti ottimi risultati, infatti i tempi di computazione rimangono lineari fino all'utilizzo di 4 processori per poi aumentare drasticamente. In entrambi i casi si puó dire che, in termini di tempi computazionali, la risoluzione del problema viene svolta al meglio da 4 processori mentre l'efficienza puó essere attribuita all'utilizzo di 2 processori.

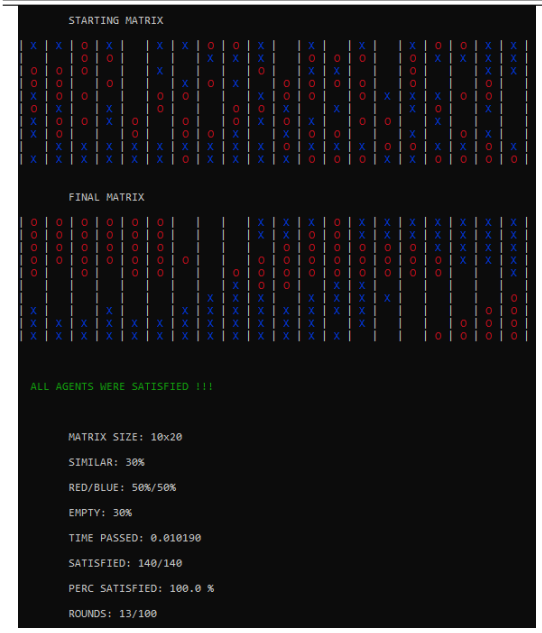
5 Correttezza

Per verificare la correttezza del codice la soluzione migliore sarebbe quella di valutare le operazioni che vengono effettuate sui singoli agenti della matrice da parte di ognuno dei processi ma, siccome viene utilizzato un generatore pseudocasuale di valori per le posizioni degli agenti insoddisfatti, é impossibile prevedere gli spostamenti che vengono fatti. Per questo motivo, la soluzione piú valida potrebbe essere quella di dimostrare che due esecuzioni del codice con funzione di randomizzazione settata su un seed predefinito a paritá di processi, generano lo stesso output, quindi la stessa matrice risultante.
Di seguito é riportato il test su 2, 4 e 6 processi con seed della funzione di randomizzazione fissato a 1, taglia matrice 40x40 (30% celle vuote, restante parte divisa in parti uguali tra valori X e valori O) e percentuale di soddisfazione al 30%.

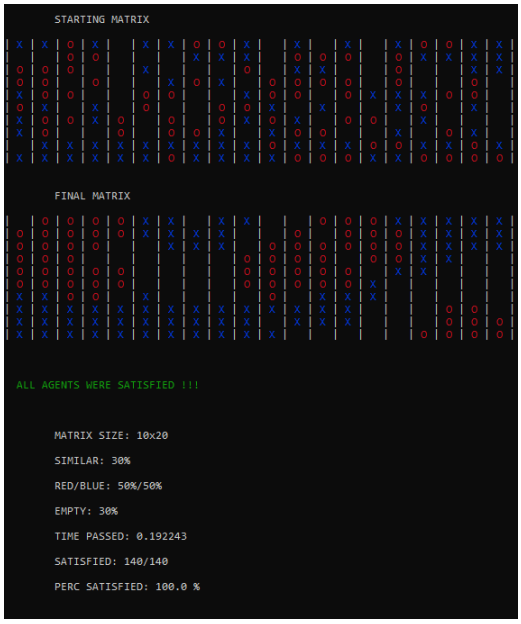
5.1 2 processi

Test 1	Test 2
	

5.2 4 processi

Test 1	Test 2
	

5.3 6 processi

Test 1	Test 2
	

6 Conclusioni

In conclusione si puó affermare che la risoluzione del problema con l'utilizzo della programmazione distribuita ha portato ad ottimi risultati che sottolineano ancora il fatto di quanto le prestazioni possano migliorare con l'uso di un determinato numero di processori grazie alla distribuzione del carico di lavoro. Inoltre, l'implementazione di un modello coerente con quello ideato da T. Schelling ha permesso di capire ancor di piú gli intenti e le finalitá dell'economista statunitense.