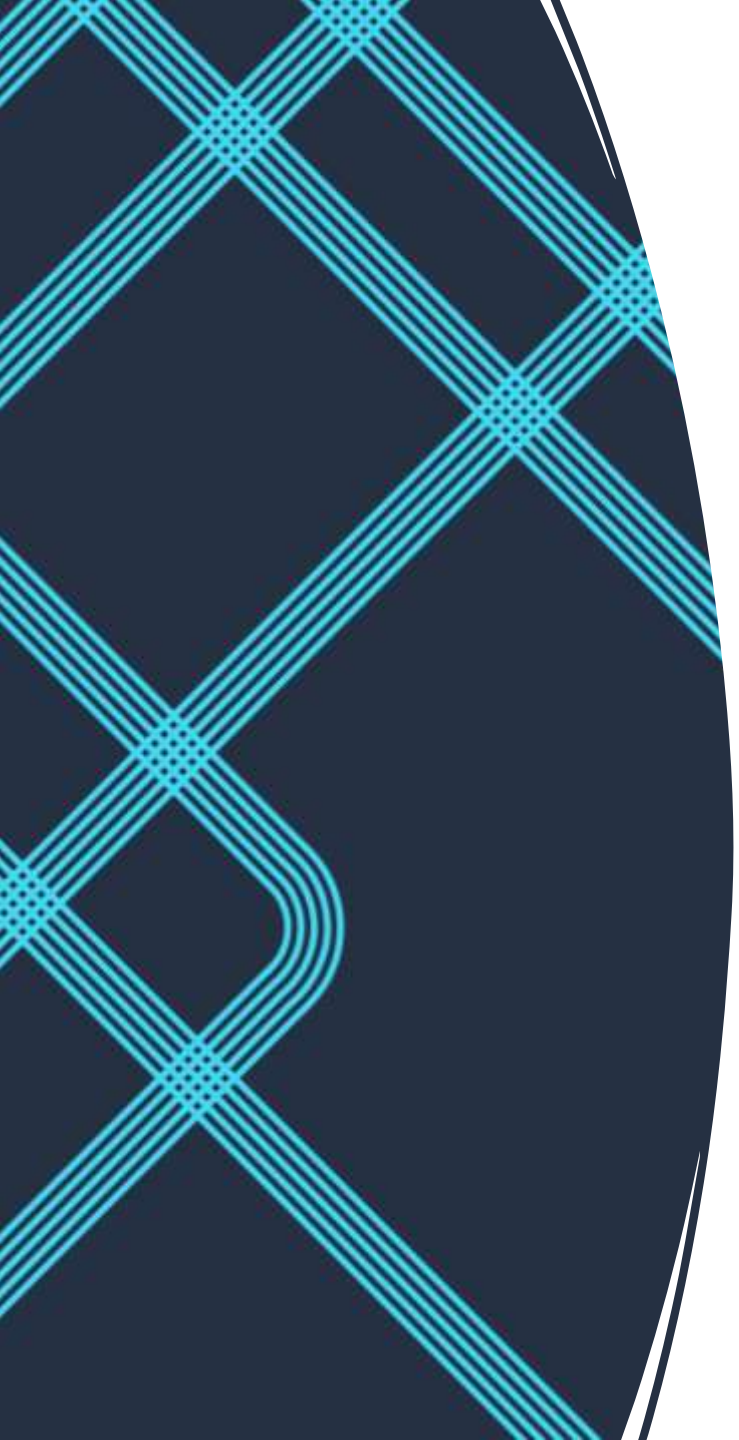


# Progetto APSD

Matteo Canino

Pierfrancesco Napoli

Witon Sebastian

# Implementazione dell'Automa Cellulare

- **Inizializzazione dei parametri e caricamento da file**
- **Partizionamento dei dati**
- **Scambio Bordi**
- **Stampa**
- **Funzione di Transizione e thread pool**
- Performance
- Allegro

# Inizializzazione dei parametri e caricamento da file

Per la lettura dei dati da file è stata utilizzata la funzione init()

- **loadConfiguration()** carica i dati dal file di configurazione
- **loadBigM()** carica i dati dal file di Input nella matrice bigM. Questo processo viene effettuato solamente dal Rank 0

```
void init(){ //Carcamento parametri da file di configurazione
    loadConfiguration();
    if(Rank==0)
        loadBigM(); //Rank 0 legge la matrice globale
}
void loadConfiguration(){ //Carcamento parametri da file di configurazione
    std::ifstream configurazione("Configuration.txt");
    if(configurazione.is_open()){
        configurazione >> xPartitions >> yPartitions >> nThreads >> steps;
        configurazione.close();
    }else{
        printf("Error opening configuration file!");
        exit(1);
    }
}
void loadBigM(){ //Carcamento matrice globale da file
        bigM = new int[NROWS*NCOLS];
        std::ifstream Input("Input.txt");
        if(Input.is_open()){
            char c;
            int i=0;
            while (Input.get(c)){
                if(c!='0' && c!='1')
                    continue;
                bigM[i]=c - '0';
                i++;
        }
            Input.close();
        }else{
            printf("Error opening Input file!");
            exit(1);
    }
}
```

# Inizializzazione delle matrici

la matrice del gioco della vita va linearizzata in questo modo potremmo lavorare su memoria contigua , avere una gestione migliore di chache e heap e sarà più facile spezzare i dati.

- **2 righe e 2 colonne** in più per le halo cell
- **la macro v(r,c)** permette l'accesso alle matrici readM e writeM
- **la macro h(r,c**) permette l'accesso alla matrice bigM

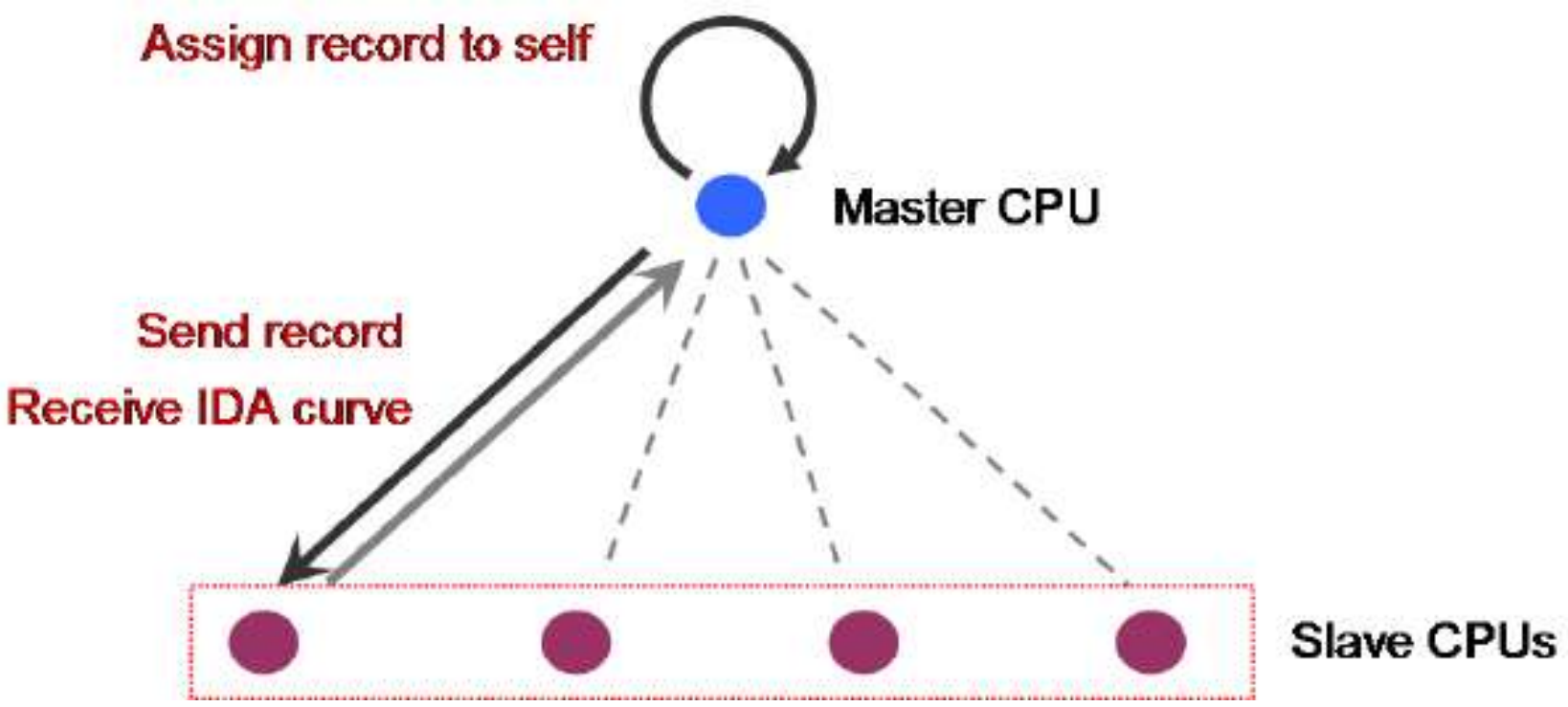
```
readM = new int[(NROWS/yPartitions+2)*(NCOLS/xPartitions+2)];
writeM = new int[(NROWS/yPartitions+2)*(NCOLS/xPartitions+2)];

        //macro per accedere alla matrice locale
        #define v(r,c) ((r)*(NCOLS/xPartitions+2)+(c))

        //macro per accedere alla matrice globale
        #define h(r,c) ((r)*(NCOLS)+(c))
```

# Partizionamento dei Dati

- Il Partizionamento dell'input da file viene effettuato come un Master/Slave democratico, cioè anche il Master prende una parte di input.
- La gestione del load balancing è statica
- La tecnica di Mapping è Block



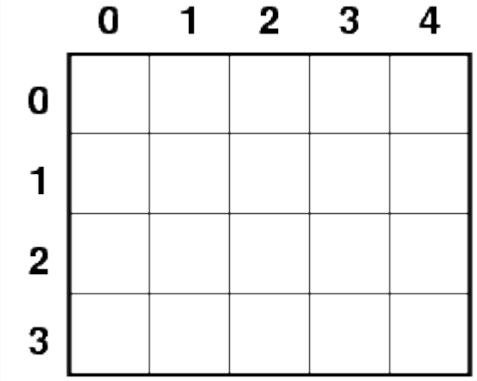
```
void initAutoma(){
    if(Rank==0){  //Rank 0 prende la prima parte della matrice globale
        int dest=1;
        for(int i=0; i<NROWS/yPartitions+2; i++){
            for(int j=0; j<NCOLS/xPartitions+2; j++){
                if(i==0 || i==NROWS/yPartitions+1 || j==0 || j==NCOLS/xPartitions+1)
                    readM[v(i,j)]=0;
                else{
                    readM[v(i,j)]=bigM[h(i-1,j-1)];
                    }
            }
        }
        for(int i=0; i<yPartitions; i++){  //invia le altre porzioni
            for(int j=0; j<xPartitions; j++){
                if(i==0 && j==0){
                    continue;
                }else{
                    MPI_Send(&bigM[h(i*(NROWS/yPartitions),j*(NCOLS/xPartitions))], 1, bigMtype, dest, 0, MPI_COMM_WORLD);
                dest++;}
             }
        }
    }else{  //Gli altri rank ricevono la porzione di matrice globale
        MPI_Status stat;
        for(int i=0; i<NROWS/yPartitions+2; i++){
            for(int j=0; j<NCOLS/xPartitions+2; j++){
                readM[v(i,j)]=0;
            }
        }
        MPI_Recv(&readM[v(1,1)], 1, rec, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    }
}
```

```
//salvataggio dei dati da matrice locale a globale
MPI_Type_vector(NROWS/yPartitions, NCOLS/xPartitions, (NCOLS/xPartitions)*xPartitions, MPI_INT, &bigMtype);
MPI_Type_commit(&bigMtype);
//salvataggio dei dati da matrice globale a locale
MPI_Type_vector(NROWS/yPartitions, NCOLS/xPartitions, (NCOLS/xPartitions)+2, MPI_INT, &rec);
MPI_Type_commit(&rec);
```

# Datatype derivati utilizzati

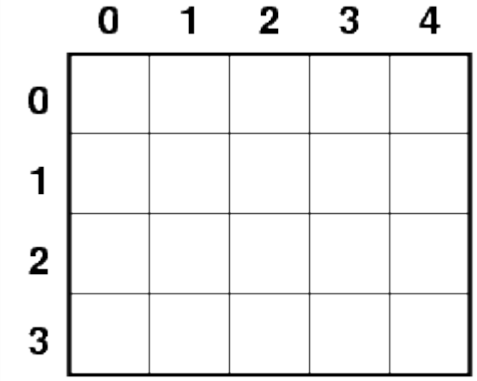
- **bigMtype**: permette di inviare una porzione “quadrata” o “rettangolare” della matrice globale. Che sarà la porzione di un worker

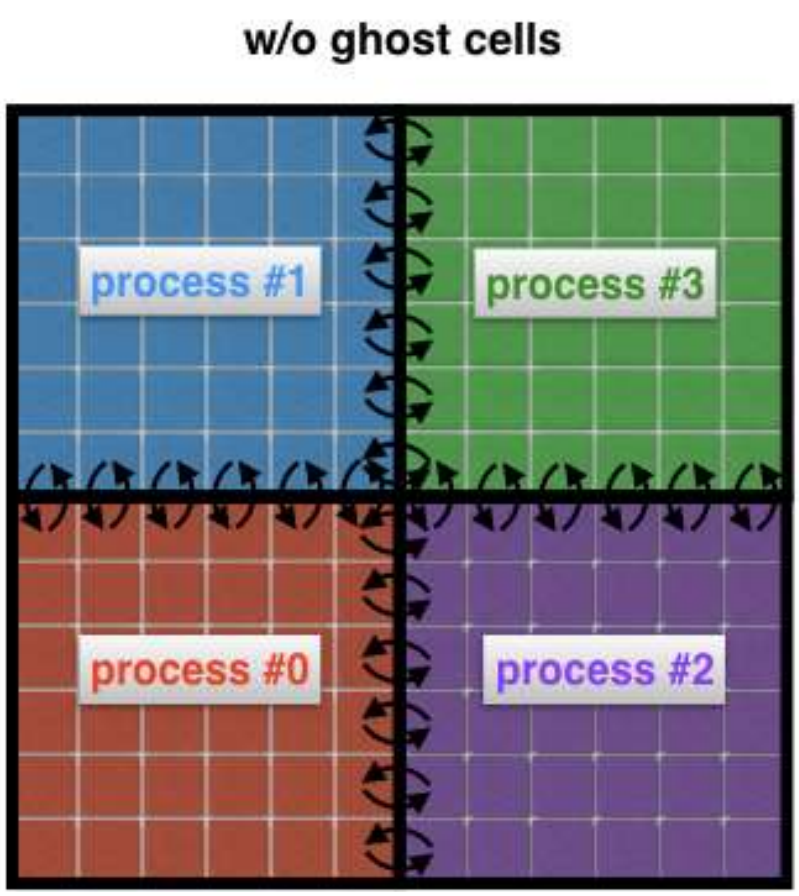
esempio:

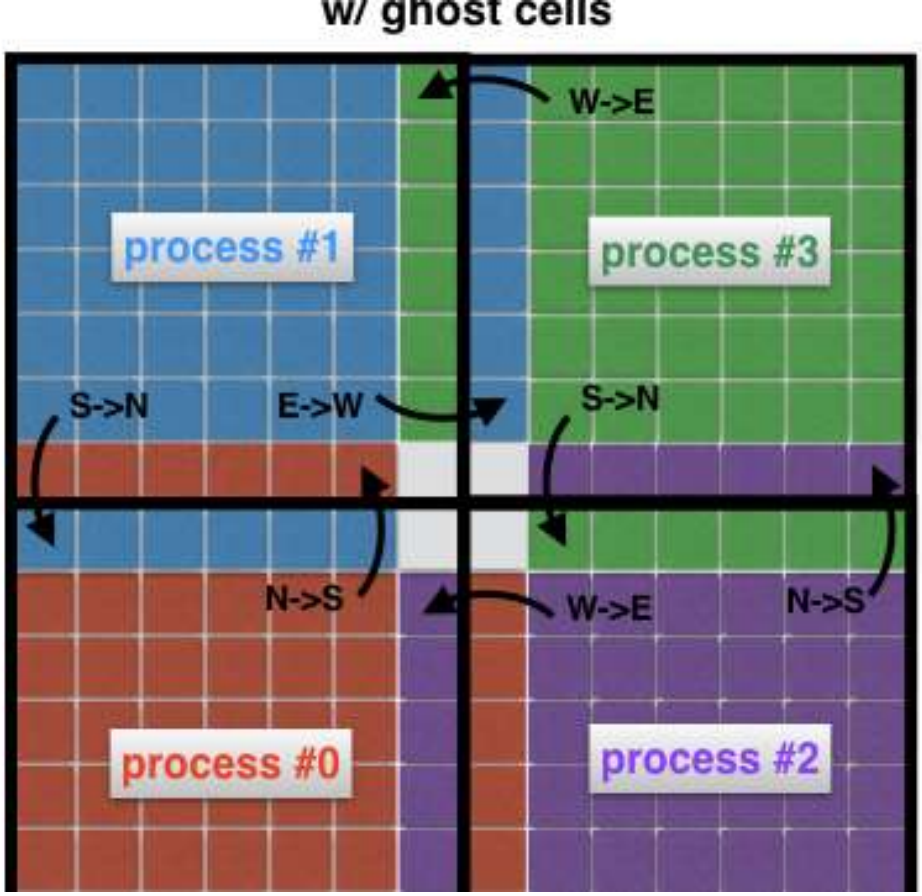



- **rec**: Il processo riceve la sua porzione di matrice, nel salvarla bisogna tener conto delle halo cell

esempio:






# Scambio dei Bordi

- Ogni matrice locale viene circondata con uno stato di celle chiamate halo cell.
- Ad ogni stato, un processo sincronizza lo stato delle halo scambiando dati con il processo vicino
- In questo caso viene utilizzata la messaggistica safe poichè I dati di righe e colonne sono dipendenti fra di loro

```
void exchBoard(){   //Scambio bordi fra vicini
    MPI_Request request;
    MPI_Status status;
    int c;
    //colonne
    MPI_Send(&readM[v(0,NCOLS/xPartitions)], 1, columnType, rankRight, 17, MPI_COMM_WORLD);
    MPI_Send(&readM[v(0,1)], 1, columnType, rankLeft, 20, MPI_COMM_WORLD);
    MPI_Recv(&readM[v(0,0)], 1, columnType, rankLeft, 17, MPI_COMM_WORLD, &status);
    MPI_Recv(&readM[v(0,NCOLS/xPartitions+1)], 1, columnType, rankRight, 20, MPI_COMM_WORLD, &status);

    //righe
    MPI_Send(&readM[v(NROWS/yPartitions,0)], NCOLS/xPartitions+2, MPI_INT, rankDown, 12, MPI_COMM_WORLD);
    MPI_Send(&readM[v(1,0)], NCOLS/xPartitions+2, MPI_INT, rankUp, 15, MPI_COMM_WORLD);
    MPI_Recv(&readM[v(NROWS/yPartitions+1,0)], NCOLS/xPartitions+2, MPI_INT, rankDown, 15, MPI_COMM_WORLD, &status);
    MPI_Recv(&readM[v(0,0)], NCOLS/xPartitions+2, MPI_INT, rankUp, 12, MPI_COMM_WORLD, &status);
}
```

```
//invio e ricezione di una colonna in matrici locali
MPI_Type_vector(NROWS/yPartitions+2, 1, NCOLS/xPartitions+2, MPI_INT, &columnType);
MPI_Type_commit(&columnType);
```

# Stampa

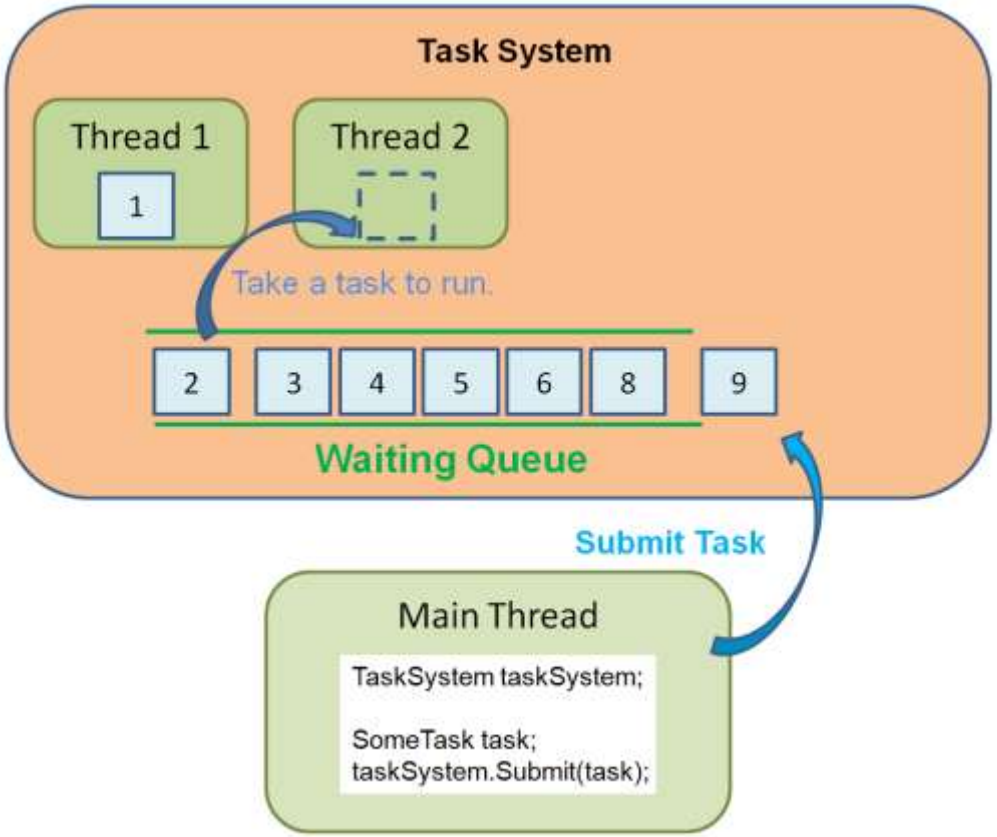
La stampa viene effettuata dal Master dopo che ha ricevuto tutte le matrici dai worker

Ritornano utili I due datatype rec e bigMtype

```
void print(int step){
    //I processi mandano la loro porzione al processo 0 per stampare
    if(Rank!=0){
        MPI_Send(&readM[v(1,1)], 1, rec, 0, 29, MPI_COMM_WORLD);
    }
    else {
        int dest=1;
        MPI_Status stat;
        printf("Step:  %d \n",step+1);
        for(int i=0; i<yPartitions; i++){
            for(int j=0; j<xPartitions; j++){
                if(i==0 && j==0){
                    for(int c=1; c<NROWS/yPartitions+1; c++){
                        for(int r=1; r<NCOLS/xPartitions+1; r++){
                            bigM[h(c-1,r-1)]=readM[v(c,r)];
                        }
                    }
                }else{
                    MPI_Recv(&bigM[h(i*(NROWS/yPartitions),j*(NCOLS/xPartitions))], 1, bigMtype, dest, 29, MPI_COMM_WORLD, &stat);
                    dest++;
                }
            }
        }
    }
```

# Funzione di transizione e thread pool

- Per la funzione di transizione viene utilizzata la soluzione della **Pool of Task.**
- Il Master crea le task e le aggiunge alla fine della coda
- Un worker appena libero esegue la task al'inizio della coda
- Il load balancing è dinamico
- Il Master è ogni singolo processo MPI e I worker sono i pthread
- Master/Slave non collaborativo



## Coda e oggetto cell

```
//Gestione pthread con thread pool
struct cell{  // Struttura per la coda di celle da calcolare
    cell(int i, int j):i(i), j(j){}
    cell(){}
    int i;
    int j;
};
std::queue<cell> q;
```

## Aggiunta di elementi nella coda

```
void transFunc(){   //funzione di aggiunta task alla thread pool
    for(int i=1;i<NROWS/yPartitions+1;i++){
        for(int j=1;j<NCOLS/xPartitions+1;j++){
            cell c(i,j);
            q.push(c);
            pthread_cond_broadcast(&cond);
            }}
            pthread_create(&threadWait, NULL, &runWait, NULL);
}
```

```
void transitionFunction(int x, int y){  //Funzione di transizione
    int cont=0;// Conto i vicini vivi
    for(int di=-1;di<2;di++)
        for(int dj=-1;dj<2;dj++)
            if ((di!=0 || dj!=0) &&readM[v((x+di+NROWS)%NROWS,(y+dj
                cont++;
    // Regole Gioco della Vita
    pthread_mutex_lock(&mutex);
    if (readM[v(x,y)]==1)
        if (cont==2 || cont ==3)
            writeM[v(x,y)]=1;
        else
            writeM[v(x,y)]=0;
    else
        if (cont ==3)
            writeM[v(x,y)]=1;
        else
            writeM[v(x,y)]=0;
    pthread_mutex_unlock(&mutex);
```

Funzione di transizione per un cella (i,j)

```
void * run(void * arg){ //thread function, ogni thre
    while( !ending){
        if(q.empty())
            pthread_cond_signal(&condWait);
        pthread_mutex_lock(&mutex);
    while (q.empty() && !ending)
        pthread_cond_wait(&cond, &mutex);
    if(!q.empty()){
        cell c=q.front();
        q.pop();
        pthread_mutex_unlock(&mutex);
        transitionFunction(c.i, c.j);
        }else{
            pthread_mutex_unlock(&mutex);
            }}
    return NULL;
}
```

Rimozione di un elemento dalla coda

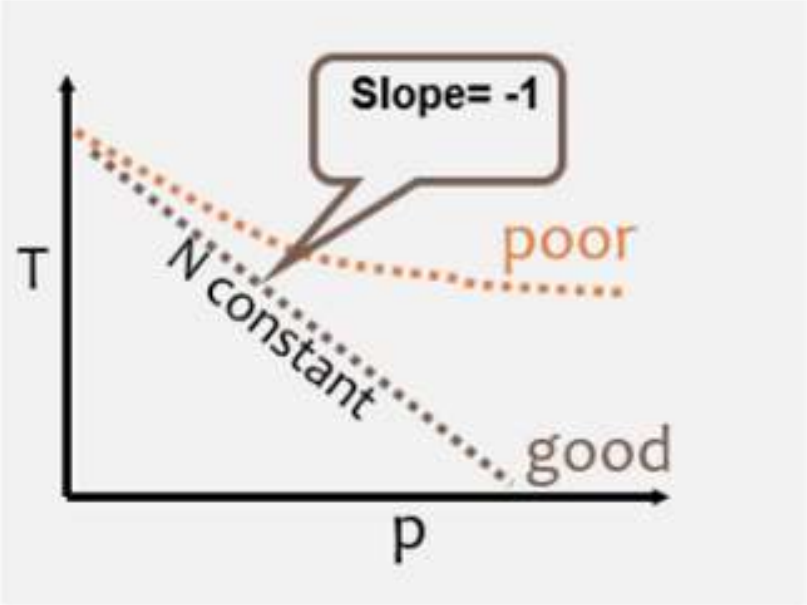
# Performance

Avendo una matrice 8x20 il tempo di esecuzione parallelo è:

- con 4 processi MPI: 1 s
- con 8 processi MPI: 0.65 s

Da questo notiamo che:

- quando il numero di processori raddoppia da 4 a 8, il tempo di esecuzione si riduce approssimativamente della metà, il che suggerisce una **scalabilità forte**.
- Se definiamo ts=1, lo **speedup** con 4 processori è 1/1 = 1 e con 8 processori è 1/0.65 = 1.54.
- **L'efficienza** con 4 processori è 1/4 = 0.25 e con 8 processori è 1.54/8 = 0.19. Quindi come ci aspettavamo l'efficienza diminuisce all'aumentare del numero di



# Allegro

Allegro è una libreria open source per la creazione di videogiochi. Sviluppata in C, fornisce delle funzioni per la gestione della grafica 2D, manipolazione delle immagini, stampa di testo a schermo, riproduzione audio, lettura degli input e timers. Il nome è un acronimo ricorsivo di **A**llegro **L**ow **LE**vel **G**ame **RO**utines.

# Variabili Allegro

Il nostro progetto fa uso della libreria Allegro per poter visualizzare l'andamento di esecuzione dell'automa cellulare istante per istante.

Come prima cosa abbiamo scaricato la libreria Allegro e l'abbiamo inclusa all'interno del nostro progetto mediante la seguente istruzione:

```
#include <allegro.h>
```

Successivamente abbiamo creato tre variabili globali utili per l'utilizzo della libreria Allegro:

1. La prima variabile (BITMAP *buffer;) è la finestra in cui vado a disegnare.
2. Le altre due variabili (int nero, bianco;) rappresentano invece i colori che andiamo ad usare per disegnare in Allegro. Useremo, infatti, il nero per lo sfondo e il bianco per disegnare le celle.

Le variabili sono, quindi, le seguenti:

```
BITMAP *buffer;
int nero, bianco;
```

# Funzioni Allegro

La gestione del programma Allegro fa uso di due funzioni:

1. La prima funzione, initAllegro(), non richiede nessun parametro e serve per inizializzare la libreria Allegro.

```
void initAllegro();
```

Siccome il processo con rank uguale a 0 possiede le informazioni relative all'intera matrice facciamo inizializzare Allegro proprio al rank 0 così che lui possa stampare a video l'intera matrice e quindi l'andamento istante per istante dell'intero automa cellulare.

```
if(Rank == 0)
    initAllegro();
```

La funzione viene chiamata nel main dopo aver inizializzato le strutture dati (readM e writeM) con i dati presi da file.

# Funzioni Allegro

2. La seconda funzione, drawWithAllegro(int step), richiede come parametro il numero di step per andarlo poi a stampare a video insieme all'andamento dell'automa cellulare.

```
void drawWithAllegro(int step);
```

La funzione viene chiamata all'interno della funzione «print» andando a fare una duplice stampa. Infatti viene stampata l'intera matrice a video sia sul terminale e sia tramite visualizzatore allegro.

```
if(Rank==0){
    //disegno con allegro sul rank 0
    drawWithAllegro(step);

    //stampo sul terminale senza allegro
    for(int i=0; i<NROWS; i++){
        for(int j=0; j<NCOLS; j++){
            printf("%d ", bigM[h(i, j)]);
        }
        printf("\n");
    }
    printf("----------\n");
}
```

**OSS**: è sempre il processo con rank uguale a 0 che stampa!

Il programma Allegro termina con l'istruzione «END_OF_MAIN();», posta alla fine del main.

# initAllegro

```
//funzione per inizializzare allegro
void initAllegro(){
    //inizializzo allegro
    allegro_init();

    //setto i colori per le altre funzioni
    set_color_depth(24);

    //creo lo schermo in cui vado a disegnare...
    buffer = create_bitmap(WIDTH, HEIGHT);
    //...e definisco di metterla nella finestra
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, WIDTH, HEIGHT, 0, 0);

    //con queste tre istruzioni vado a settare un titolo alla finestra che vado a creare
    char windowTitle[50];
    sprintf(windowTitle, "Allegro Screen");
    set_window_title(windowTitle);

    //inizializzo i colori bianco e nero
    nero = makecol(0, 0, 0);
    bianco = makecol(255, 255, 255);
}
```

# initAllegro

La funzione «initAllegro» è così composta:

- allegro_init(); = serve per inizializzare allegro.
- set_color_depth(24); = imposta il formato di pixel da utilizzare nelle chiamate successive a set_gfx_mode() e create_bitmap(). Le profondità valide sono 8, 16, 24, 32 bit.
- buffer = create_bitmap(WIDTH, HEIGHT); = crea lo schermo in cui disegnare di dimensione WIDTH*HEIGHT (due costanti definite all'inizio del programma).
- set_gfx_mode(GFX_AUTODETECT_WINDOWED, WIDTH, HEIGHT, 0, 0); = serve per impostare la modalità grafica del sistema. Accetta diversi parametri che specificano la modalità grafica desiderata, come la risoluzione dello schermo, il numero di bit per pixel (profondità del colore), la frequenza di aggiornamento, ecc.

# initAllegro

- char windowTitle[50]; = creo un array di caratteri utile per contenere il titolo della schermata allegro.
- sprintf(windowTitle, «Allegro Screen»); = funzione che formatta la stringa «Allegro Screen» e la salva nel buffer windowsTitle.
- set_window_tile(windowTitle); = imposta il titolo della finestra dell'applicazione Allegro.
- nero = makecol(0, 0, 0); = serve per creare un colore personalizzato, nel caso specifico il colore nero, utilizzando i valori dei componenti RGB.
- bianco = makecol(255, 255, 255); = serve per creare un colore personalizzato, nel caso specifico il colore bianco, utilizzando i valori dei componenti RGB.

# drawWithAllegro

```
//Funzione per disegnare
void drawWithAllegro(int step){
    //NOTA: questa funzione si adatta alle varie dimensioni di input ricalcolando sempre altezza e larghezza di ogni blocco in base al numero di righe e colonne

    //qua vado a calcolare larghezza e altezza di ogni singolo quadrato che vado a disegnare
    int const CELL_WIDTH = WIDTH / NCOLS;
    int const CELL_HEIGHT = HEIGHT / NROWS;

    //qui faccio partire un doppio ciclo for per scorrere la matrice e disegnare.
    for (int i = 0; i < NROWS; i++)
        for (int j = 0; j < NCOLS; j++){
            //calcolo la x e la y iniziale di ogni blocco che devo disegnare
            int x = i * CELL_HEIGHT;
            int y = j * CELL_WIDTH;

            //switch per verificare contenuto della cella della matrice
            switch (bigM[h(i, j)]) {
                case 0:
                    //se la cella (i, j) è 0 vado a disegnare un quadrato nero.
                    //parte dalla posizione x, y e si sviluppa in larghezza di y+CELL_WIDTH; si sviluppa in altezza di x+CELL_HEIGHT
                    rectfill(buffer, y, x, y + CELL_WIDTH, x + CELL_HEIGHT, nero);
                    break;
                case 1:
                    //se la cella (i, j) è 1 vado a disegnare un quadrato bianco.
                    //parte dalla posizione x, y e si sviluppa in larghezza di y+CELL_WIDTH; si sviluppa in altezza di x+CELL_HEIGHT
                    rectfill(buffer, y, x, y + CELL_WIDTH, x + CELL_HEIGHT, bianco);
                    break;
            }
        }
```

# drawWithAllegro

Nella funzione «drawWithAllegro» andiamo come prima cosa a calcolare altezza (HEIGHT / NROWS) e larghezza (WIDTH / NCOLS) di ogni singola cella.
Successivamente andiamo, attraverso un doppio ciclo for, a scorrere l'intera matrice che contiene lo stato attuale dell'automa. Attraverso un costrutto switch andiamo a verificare se la cella attuale contiene 0 o 1:

- Se la cella attuale contiene valore 0, vuol dire che l'automa non si trova in quella cella. Andiamo, quindi, a colorare la cella con un rettangolo nero. La funzione che ci permette di effettuare ciò è: rectfill(buffer, y, x, y + CELL_WIDTH, x + CELL_HEIGHT, nero);
  Questa funzione disegna un rettangolo riempito nel buffer grafico specificato, utilizzando le coordinate di inizio e fine del rettangolo. Nel codice fornito, il rettangolo viene disegnato dalla posizione (x, y) alla posizione (y + CELL_WIDTH, x + CELL_HEIGHT) e viene riempito col colore nero.

# drawWithAllegro

- Se la cella attuale contiene valore 1, vuol dire che l'automa si trova in quella cella. Andiamo, quindi, a colorare la cella con un rettangolo bianco. La funzione che ci permette di effettuare ciò è: rectfill(buffer, y, x, y + CELL_WIDTH, x + CELL_HEIGHT, bianco);

Abbiamo poi altre due istruzioni importanti:

- textprintf_ex(buffer, font, 0, 0, bianco, nero, "Step: %d", step); = viene utilizzata per disegnare del testo formattato nel buffer grafico specificato, utilizzando il font, le coordinate, i colori e la stringa di formato forniti. La funzione sostituirà il segnaposto "%d" nella stringa di formato con il valore di "step".
- blit(buffer, screen, 0, 0, 0, 0, WIDTH, HEIGHT); = viene utilizzata per copiare il contenuto di un buffer grafico sullo schermo, consentendo la visualizzazione dell'immagine o della scena all'utente.

# Risultato