Desenvolvimento de solução paralela com **MPI**
Paradigmas da Computação Paralela

Daniel Carvalho pg25302
Luís Caseiro pg27757

18 de Novembro de 2014

Conteúdo

1 Introdução **2**
1.1 Objectivos 2
1.2 **MPI** 2
1.3 Contextualização do problema 2

2 Desenvolvimento da solução paralela **2**
2.1 Versão Sequencial 2
2.2 Implementação 3
2.2.1 Topologia 3
2.2.2 Estruturas utilizadas 4
2.2.3 Função *SetLand* 4
2.2.4 Função *FillBorders* 5
2.2.5 Calcular evolução das populações 6
2.2.6 Calcular número total de cada população 6

3 Método e Resultados **6**
3.1 Método 6
3.2 Resultados 7

4 Conclusões **9**

Lista de Figuras

1 Matriz 9 X 9 dividida por 3 processos 3
2 Relação entre tamanho das matrizes e tempo de execução 7
3 Relação entre tamanho das matrizes e tempo de execução 7
4 Relação entre tamanho das matrizes e tempo de execução 8
5 Relação entre tamanho das matrizes e tempo de execução 8
6 Relação entre tamanho das matrizes e tempo de execução 9

1 Introdução

1.1 Objectivos

Com a solução a desenvolver para este problema pretende-se analisar os ganhos da computação paralela relativamente à computação sequencial, usando para isso MPI e recorrendo a um número variável de processos de forma a perceber-se a variação dos ganhos computacionais. Para isso será mostrada a abordagem tomada para resolucionar o problema, assim como o uso do MPI para paralelizar essa mesma abordagem.

1.2 MPI

De forma a ser possível paralelizar a computação deste problema recorrer-se-á ao uso do MPI (*Message Passing Interface*), que permite a passagem de informação entre os vários processos.

1.3 Contextualização do problema

A ideia geral tomar como base uma versão sequencial já realizada de um problema da classe ECO que irá servir de plataforma para a crição de um programa paralelo com funcionalidade semelhante à da versão sequencial. Tomemos como exemplo um território com uma área rectangular de terreno habitado por raposas e por coelhos. Ao longo dos anos sucessivas gerações daqueles animais são estimadas de acordo com um modelo simples de predador-presa para determinar a forma como as duas populações evoluem conjuntamente. A solução do problema parte da divisão do terreno em quadrículas com 1 quilmetro de lado e a fixação das populações iniciais. Em cada período de um ano o número de coelhos e de raposas são estimados de acordo com uma fórmula matemática discreta que representa a varição das populações em cada quadrícula como função do nascimento, da morte ou da migração para outras quadrículas.

2 Desenvolvimento da solução paralela

Antes de se começar a implementar a versão paralela paralela com **MPI**, analisou-se a versão sequencial que fora fornecida.

2.1 Versão Sequencial

A grande diferença entre a versão sequencial e a versão paralela que irá ser implementada, é que quando se programa de forma paralela tira-se recurso de diversos processos/threads, enquanto que programando sequencialmente apenas se tem um processo. Tendo um processo, criam-se duas matrizes para representar a população atual de coelhos e raposas, e mais duas matrizes para guardar as previsões para os proximos anos. Como existe apenas um processo a executar, todas as funcões que atuam sobre estas matrizes têm todos os dados que necessitam, não precisando de receber dados que não teriam disponiveis.

2.2 Implementação

Depois de se analisar a versão sequencial fornecida, começou-se a desenvolver a versão paralela utilizando **MPI**. Na versão paralela todos os processos irão ter matrizes locais e irão comunicar entre si de forma a cumprir os requesitos necessários para encontrar a solução.

2.2.1 Topologia

Como a estrutura com que iriamos trabalhar eram matrizes decidiu-se adoptar uma topologia cartesiana, em que o **MPI** oferece duas funções que permitem facilitar o problema. A primeira é a função *MPI_Cart_create* que permite criar-se esta topologia através da topologia base do **MPI** tendo como parâmetros o numero de dimensões e um apontador *MPI Comm* onde será guardada a nova topologia.

A segunda função é a função *MPI_Cart_shift* que permite a cada processo saber quais processos se encontram na sua vizinhança, por exemplo na figura 1 sendo a matriz com cor vermelha a area atribuída ao processo 0, o resultado desta função iria determinar que o processo 1, a verde, é vizinho do processo 0.

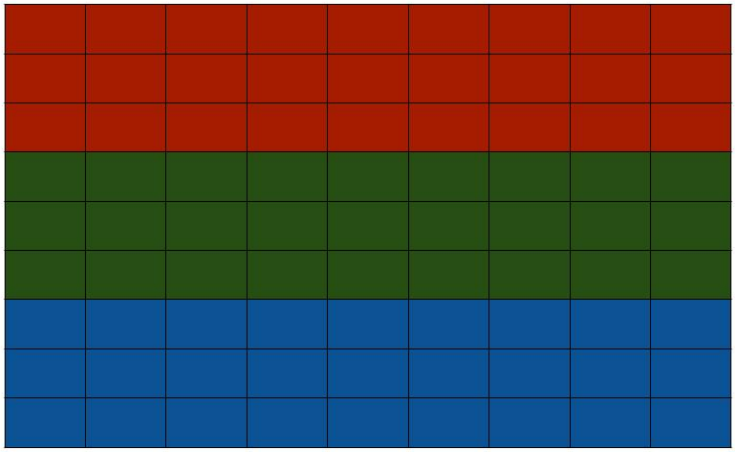

Figura 1: Matriz 9 X 9 dividida por 3 processos

Decidiu-se partir a topologia apenas por linhas de forma diminuir o número de comunicações entre processos e também manter o algoritmo mais simples, um exemplo da topologia utilizada é a figura 1.

Em seguida vem como se especificou a topologia.

```
MPI_Comm my_grid;
dim[0] = 1;
dim[1] = comm_size;
period[0]=TRUE;
period[1]=TRUE;
reorder=FALSE;
```

```
MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&my_grid);
.
.
MPI_Cart_shift(my_grid,1,1,&up,&down);
```

2.2.2 Estruturas utilizadas

De forma a utilizar mais eficientemente os recursos disponibilizados optou-se por definir as matrizes locais de cada processo, arrays unidimensionais, isto de forma a que todos os elementos da matriz estejam contiguos em memória, utilizando-se a localidade da *cache* de forma a melhorar o desempenho do programa. O acesso às posições da matriz é feito através de uma função auxiliar que converte posições bi-dimensionais em unidimensionais.

O tamanho das matrizes locais de cada processo irá depender do número de processos com que se executa o programa, ou seja, o número de linhas que cada matriz será o resultado da divisão entre o número de linhas total e o número de processos, sendo que ao processo com *rank* maior irá ser somado o resto da operação, podendo este ficar com uma matriz maior.

```
line_offset = (NS_Size)/comm_size;

if(rank == comm_size -1)
{
    line_offset = line_offset + (int)NS_Size % comm_size;
}

float* lmRabbit = malloc(line_offset*WE_Size*sizeof(float));
float* lmFox = malloc(line_offset*WE_Size*sizeof(float));

float* lmTRabbit = malloc(line_offset*WE_Size*sizeof(float));
float* lmTFox = malloc(line_offset*WE_Size*sizeof(float));
```

Definiu-se também um tipo derivado do **MPI**, de forma a reduzir o tamanho dos dados transmitidos entre processos. A estrutura será contigua em memória e irá ter o tamanho de uma linha da matriz.

```
MPI_Datatype row;
MPI_Type_contiguous((WE_Size), MPI_FLOAT, &row);
MPI_Type_commit(&row);
```

2.2.3 Função *SetLand*

A função *setLand* irá preencher as matrizes locais de cada processo, através dos índices bi-dimensionais de cada matriz local, este ponto irá levar a um resultado diferente da versão sequencial porque os limites de cada matriz local será inferior ao da matriz total que é utilizada na versão sequencial. A primeira

e última coluna das matrizes locais não será preenchida, assim como a primeira linha do processo 0 e a última do processo com *rank* maior, pois estas linhas são linhas fronteira que irão ser preenchidas em outra função 2.2.4. A codificação desta função encontra-se em anexo.

2.2.4 Função *FillBorders*

A função *FillBorders* irá preencher as fronteiras de cada matriz local, ou seja, irá preencher as posições não preenchidas pela função *setLand* 2.2.3.

Esta função terá três fases, sendo estas, preencher a primeira linha da matriz local do processo 0, preencher última linha da matriz local do processo de *rank* maior e preencher a primeira e última coluna da matriz local de cada processo. A primeira linha da matriz local do processo 0 terá de ser igual à penúltima linha do processo com *rank* maior e a última linha do processo com *rank* maior será igual à segunda linha do processo 0, ou seja, terá de existir comunicação entre estes dois processos em que cada um envia a linha que o outro necessita, abaixo está como foi feita a comunicação utilizando o tipo derivado **MPI**.

```
if(rank == 0)
{
       for(j=0; j <WE_Size;j++)
       {
           sendUp_rowR[j] = lmRabbit[getPos(1,j)];
       }

       MPI_Isend(sendUp_rowR, 1, row, comm_size - 1, 0, my_grid,reqR);

       MPI_Recv(recvDown_rowR,1, row, comm_size - 1, 0, my_grid, statusR);
       MPI_Wait(reqR,statusR);
}


if(rank == comm_size-1)
{
       for(j=0; j <WE_Size;j++)
       {
           sendDown_rowR[j] = lmRabbit[getPos(line_offset-1,j)];
       }

       MPI_Isend(sendDown_rowR, 1, row, 0, 0, my_grid,reqR);

       MPI_Recv(recvUp_rowR,1, row, 0, 0, my_grid, statusR);
       MPI_Wait(reqR,statusR);
 }
 err = FillBorder(lmRabbit,line_offset,recvUp_rowR,recvDown_rowR,rank,comm_size);
```

Em relacão ao terceiro ponto, a primeira coluna de cada matriz local será à penúltima da mesma e a última coluna será igual à segunda da mesma matriz.

Como cada processo tem todas as colunas da sua matriz não será necessário pedir colunas a outros processos. A codificação da função *FillBorders* segue em anexo.

2.2.5 Calcular evolução das populações

calcular o valor da população para anos seguintes implica utilizar valores da vizinhança, por isso será necessário existir comunicação entre processos.

Utilizaram-se assim duas funções a função *nonCriticalLines* e a função *criticalLines*. Ambas as funções irão calcular a evolução para uma posição de cada matriz local do processo em que foram chamadas, a diferença é que a função *nonCriticalLines* irá atuar em linhas em que o processo não precisa comunicar com os seus vizinhos pois todos os valores necessários estão na sua matriz local, enquanto que a função *CriticalLines* irá atuar em linhas em que o processo precisa de comunicar com os seus vizinhos, existindo trocas de dados entre estes. O motivo de separar esta função foi que enquanto o processo pode calcular o valor das linhas não críticas enquanto espera pelos dados para calcular o valor das linhas críticas. A codificação destas funções encontra-se em anexo.

2.2.6 Calcular número total de cada população

Para calcular o numero total de cada população depois de um determinado tempo, irá ser feito somando os valores de cada matriz local de cada processo, sendo depois aplicada a função *MPI_Reduce* que irá somar os valores das matrizes locais de todos os processos guardando o valor numa variavel global. Abaixo segue a codificação desta operação.

```
err = GetPopulation(lmTRabbit,&nbrab,line_offset,rank,comm_size);
err = GetPopulation(lmTFox,&nbfox,line_offset,rank,comm_size);
MPI_Reduce(&nbrab,&totalrab,1,MPI_FLOAT,MPI_SUM,0,my_grid);
MPI_Reduce(&nbfox,&totalfox,1,MPI_FLOAT,MPI_SUM,0,my_grid);
```

Para calcular o tempo de execução da versão paralela utilizou-se a função *MPI_Wtime*.

3 Método e Resultados

3.1 Método

Todos os resultados apresentados foram obtidos numa com processador *intel core i7-4700MQ* com frequencia de 2.4 GHZ e com tamanhos de *cache* L1 = 256 KB, *cache* L2 = 1024 KB e *cache* L3 = 6144 KB. O sistema operativo utilizado foi linux. Utilizou-se *gcc* 4.9.

3.2 Resultados

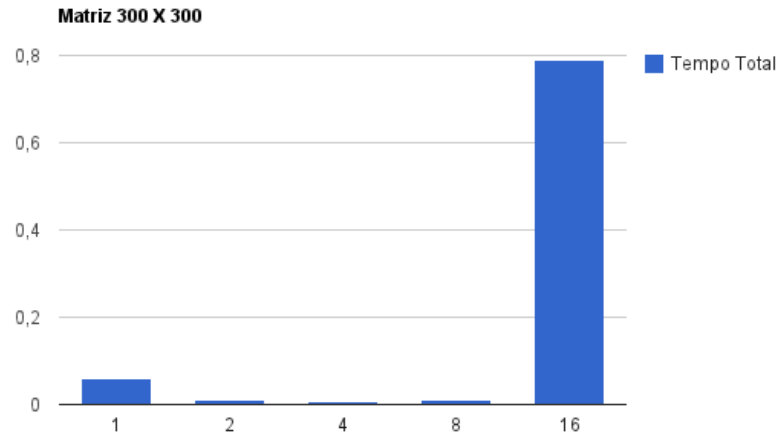

Figura 2: Relação entre tamanho das matrizes e tempo de execução

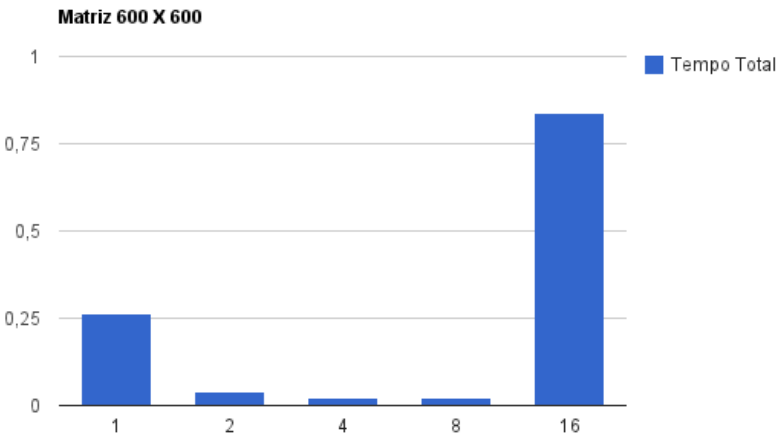

Figura 3: Relação entre tamanho das matrizes e tempo de execução

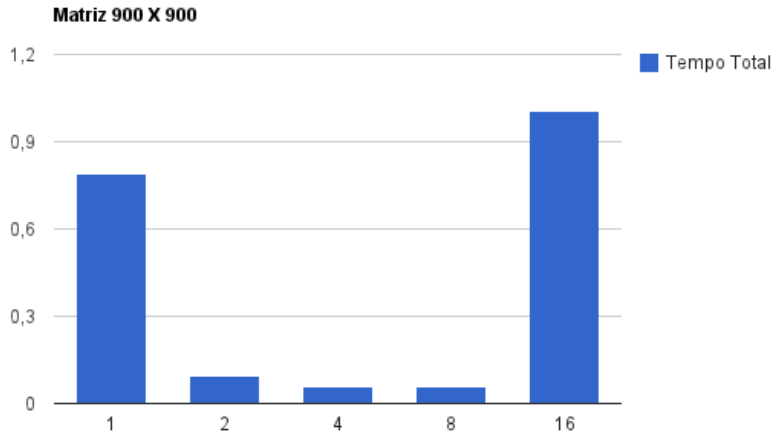

Figura 4: Relação entre tamanho das matrizes e tempo de execução

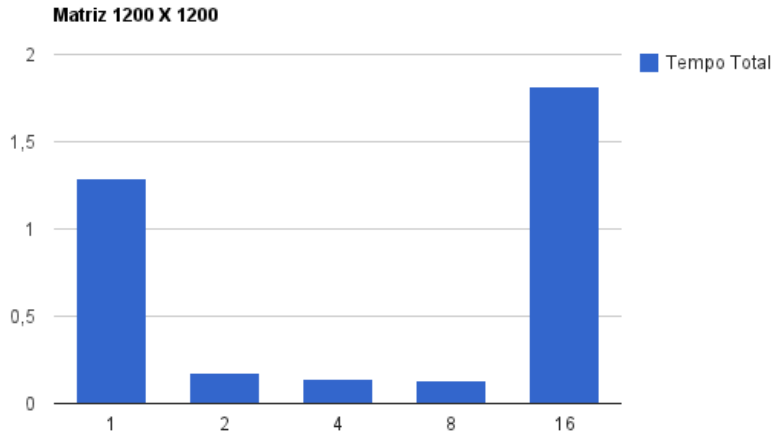

Figura 5: Relação entre tamanho das matrizes e tempo de execução

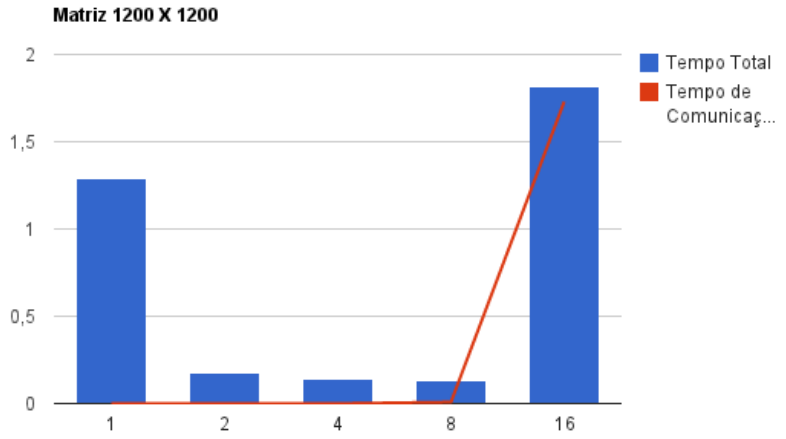

Figura 6: Relação entre tamanho das matrizes e tempo de execução

4 Conclusões

Analisando os tempos de execução conclui-se que com as versão paralelas de dois até oito processos existe uma melhoria de performance em relação à versão sequencial do programa, isto porque o problema é dividido em matrizes mais pequenas sendo estas distribuidas por diferentes processos que trabalham em paralelo em cores diferentes, enquanto que na versão paralela com dezasseis processos a performance quebra por duas razões, o tempo de comunicação é maior porque a existencia de vários processos exige um maior número de comunicações e também o hardware com que as medições foram feitas não suporta dezasseis processos a correr simultaneamente em cada *core* provocando uma queda no desempenho do programa.

Podemos ver também que o tamanho total das matrizes influência a performance e quanto maior for o tamanho destas mais este vai decair, isto porque a memória cache não conseguirá ter as matrizes completas em *cache* o que irá fazer com que o processador tenha de ir buscar valores à memória central baixando a performance do programa.

Com isto podemos concluir que uma versão paralela em **MPI** permite aumentar a performance do programa em relação à sua versão sequencial, mas tem limites que devemos ter em conta, como tempos de comunicação entre processos e limitações do hardware da máquina em que estamos a trabalhar que devemos ter em conta de forma a tirar o máximo de cada programa.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include "param.h"
#include "macro.h"

float AlR = -0.2;
float BtR = 0.6;
float MuR = 0.01;
float BtF = 0.6;
float AlF = -1.8;
float MuF = 0.02;
//char* s = "Passou";

//char* filepathLRabbit = "/home/luis/git/Echo/fileRabbit.txt";
//char* filepathLFox = "/home/luis/git/Echo/fileFox.txt";


int getPos(int i, int j)
{
    int pos = i*(WE_Size) + j;
    return pos;
}

void printMatrix(float* matrix,int line_offset,char* filepath,int rank)
{
    int i,j,pos;
    FILE* f = fopen(filepath,"w+");
    fprintf(f, "Matriz local do processo %d\n",rank );
    for (i = 0; i < line_offset; i++)
    {
        fprintf(f, "linha: %d \n", i);
        for (j = 0; j < WE_Size; j++)
        {
            pos=getPos(i,j);
            fprintf(f, " %f ", matrix[pos]);
        }
        fprintf(f, "\n" );
    }
    fclose(f);
}

void update(float* matrix1,float* matrix2,int line_offset)
{
    int i,j,pos;
    for (i = 0; i < line_offset; i++)
    {
        for (j = 0; j < WE_Size; j++)
```

```
        {
            pos=getPos(i,j);
            matrix1[pos] = matrix2[pos];
        }
    }
}

int SetLand ( float *Rabbit,float *Fox, int offset,int my_rank,int comm_sz);

int nonCriticalLines(float* Rabbit,float* Fox,float* TRabbit,float* TFox,int offset);

int criticalLines(float* lineUpR,float* lineDownR,float* lineUpF,float* lineDownF,
    float* Rabbit,float* Fox,float* TRabbit,float* TFox, int my_rank, int comm_sz,int offs

int FillBorder(float  *Animal,int line_offset, float* lineUp, float* lineDown, int rank, i

int GetPopulation(float *Animal,float *tcount,int offset,int rank, int comm_size);




int main(int argc, char *argv[])
{
    int rank,comm_size;
    MPI_Comm my_grid;
    int dim[2],period[2],reorder;
    int up,down,line_offset,err=0,k,j,pos;
    float nbrab,nbfox,totalrab,totalfox;
    MPI_Init(&argc, &argv);
    double MPI_Wtime(void);
    double start, finish;
    start=MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&comm_size);
    MPI_Request reqR[4],reqF[4];
    MPI_Status statusR[4],statusF[4];
    line_offset = (NS_Size)/comm_size;
    dim[0] = 1;
    dim[1] = comm_size;
    period[0]=TRUE;
    period[1]=TRUE;
    reorder=FALSE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&my_grid);

    if(rank == comm_size -1)
    {
        line_offset = line_offset + NS_Size % comm_size;
    }
```

```
//Criação de dados derivados
MPI_Datatype row;
MPI_Type_contiguous((WE_Size), MPI_FLOAT, &row);
MPI_Type_commit(&row);

float* sendUp_rowR = malloc(WE_Size*sizeof(float));
float* sendDown_rowR = malloc(WE_Size*sizeof(float));

float* recvUp_rowR = malloc(WE_Size*sizeof(float));
float* recvDown_rowR = malloc(WE_Size*sizeof(float));

float* sendUp_rowF = malloc(WE_Size*sizeof(float));
float* sendDown_rowF = malloc(WE_Size*sizeof(float));

float* recvUp_rowF = malloc(WE_Size*sizeof(float));
float* recvDown_rowF = malloc(WE_Size*sizeof(float));

float* lmRabbit = malloc(line_offset*WE_Size*sizeof(float));
float* lmFox = malloc(line_offset*WE_Size*sizeof(float));

float* lmTRabbit = malloc(line_offset*WE_Size*sizeof(float));
float* lmTFox = malloc(line_offset*WE_Size*sizeof(float));


err = SetLand(lmRabbit,lmFox,line_offset,rank,comm_size);

for( k=1; k<=NITER; k++)
{

    nbrab=0;
    nbfox=0;
    totalrab=0;
    totalfox=0;

    if(rank == 0)
    {
        for(j=0; j <WE_Size;j++)
        {
            sendUp_rowR[j] = lmRabbit[getPos(1,j)];
        }

        MPI_Isend(sendUp_rowR, 1, row, comm_size - 1, 0, my_grid,reqR);

        MPI_Recv(recvDown_rowR,1, row, comm_size - 1, 0, my_grid, statusR);

        for(j=0; j <WE_Size;j++)
        {
            sendUp_rowF[j] = lmFox[getPos(1,j)];
        }
```

```
    MPI_Isend(sendUp_rowF, 1, row, comm_size - 1, 0, my_grid,reqF);

    MPI_Recv(recvDown_rowF,1, row, comm_size - 1, 0, my_grid, statusF);

    MPI_Wait(reqR,statusR);
    MPI_Wait(reqF,statusF);

}

if(rank == comm_size-1)
{
    for(j=0; j <WE_Size;j++)
    {
        sendDown_rowR[j] = lmRabbit[getPos(line_offset-1,j)];
    }

    MPI_Isend(sendDown_rowR, 1, row, 0, 0, my_grid,reqR);

    MPI_Recv(recvUp_rowR,1, row, 0, 0, my_grid, statusR);

    for(j=0; j <WE_Size;j++)
    {
        sendDown_rowF[j] = lmFox[getPos(line_offset-1,j)];
    }

    MPI_Isend(sendDown_rowF, 1, row, 0, 0, my_grid,reqF);

    MPI_Recv(recvUp_rowF,1, row, 0, 0, my_grid, statusF);

    MPI_Wait(reqR,statusR);
    MPI_Wait(reqF,statusF);
}


err = FillBorder(lmRabbit,line_offset,recvUp_rowR,recvDown_rowR,rank,comm_size);
err = FillBorder(lmFox,line_offset,recvUp_rowF,recvDown_rowF,rank,comm_size);


MPI_Cart_shift(my_grid,1,1,&up,&down);

if (up == -1)
{
    down = MPI_PROC_NULL;
}
if (down == comm_size)
{
    down = MPI_PROC_NULL;
}

//////UP Lines
```

```
for(j = 0; j < (WE_Size); j++)
{
    pos=getPos(0,j);
    sendUp_rowR[j] = lmRabbit[pos];
}

MPI_Isend(sendUp_rowR, 1, row, up, 0, my_grid,reqR);


MPI_Recv(recvDown_rowR,1, row, down, 0, my_grid, statusR);

for(j = 0; j < (WE_Size); j++)
{
    pos=getPos(0,j);
    sendUp_rowF[j] = lmFox[pos];
}
MPI_Isend(sendUp_rowF, 1, row, up, 0, my_grid,reqF);

MPI_Recv(recvDown_rowF,1, row, down, 0, my_grid, statusF);

//////DOWN Lines

for(j = 0; j < (WE_Size); j++)
{
    pos=getPos(line_offset-1,j);
    sendDown_rowR[j] = lmRabbit[pos];
}
MPI_Isend(sendDown_rowR, 1, row, down, 0, my_grid,reqR);

MPI_Recv(recvUp_rowR,1, row, up, 0, my_grid, statusR);

for(j = 0; j < (WE_Size); j++)
{
    pos=getPos(line_offset-1,j);
    sendDown_rowF[j] = lmRabbit[pos];
}
MPI_Isend(sendDown_rowF, 1, row, down, 0, my_grid,reqF);

MPI_Recv(recvUp_rowF,1, row, up, 0, my_grid, statusF);

err = nonCriticalLines(lmRabbit,lmFox,lmTRabbit,lmTFox,line_offset);

MPI_Wait(reqR,statusR);
MPI_Wait(reqF,statusF);


err = criticalLines(recvUp_rowR,recvDown_rowR,recvUp_rowF,recvDown_rowF,lmRabbit,l

if( !(k % PERIOD) )
```

```
        {
            err = GetPopulation(lmTRabbit,&nbrab,line_offset,rank,comm_size);
            err = GetPopulation(lmTFox,&nbfox,line_offset,rank,comm_size);
            MPI_Reduce(&nbrab,&totalrab,1,MPI_FLOAT,MPI_SUM,0,my_grid);
            MPI_Reduce(&nbfox,&totalfox,1,MPI_FLOAT,MPI_SUM,0,my_grid);
            //if(rank == 0)
                //printf("In the year %d, the number of rabbits is %d and the number of fo
        }
        update(lmRabbit,lmTRabbit,line_offset);
        update(lmFox,lmTFox,line_offset);

    }
    free(sendUp_rowR);
    free(sendDown_rowR);
    free(sendUp_rowF);
    free(sendDown_rowF);
    free(recvUp_rowR);
    free(recvDown_rowR);
    free(recvUp_rowF);
    free(recvDown_rowF);
    free(lmRabbit);
    free(lmFox);
    free(lmTRabbit);
    free(lmTFox);

    if(rank==0)
    {
        finish=MPI_Wtime();
        printf("tempo decorrido: %f\n", finish - start);
    }
    MPI_Finalize();
    return 0;
}


int SetLand ( float *Rabbit,float *Fox, int offset,int my_rank,int comm_sz)
{
    int err,pos;
    int gi, gj;
    err = 1;
    if(my_rank==0)
    {
        for( gi=1;  gi < offset; gi++)
        {
            for( gj=1; gj<=WE_Size-2; gj++)
            {
                pos=getPos(gi,gj);
                Rabbit[pos] =
                    128.0*(gi-1)*(NS_Size-gi)*(gj-1)*(WE_Size-gj) /
                    (float)(NS_Size*NS_Size*WE_Size*WE_Size);
```

```
                Fox[pos] =
                    8.0*(gi/(float)(NS_Size)-0.5)*(gi/(float)(NS_Size)-0.5)+
                    8.0*(gj/(float)(WE_Size)-0.5)*(gj/(float)(WE_Size)-0.5);
            }
        }
    }
    if((my_rank+1) == comm_sz)
    {
        for( gi=0;  gi < (offset-1); gi++)
        {
            for( gj=1; gj<=WE_Size-2; gj++)
            {
                pos=getPos(gi,gj);
                Rabbit[pos] =
                    128.0*(gi-1)*(NS_Size-gi)*(gj-1)*(WE_Size-gj) /
                    (float)(NS_Size*NS_Size*WE_Size*WE_Size);
                Fox[pos] =
                    8.0*(gi/(float)(NS_Size)-0.5)*(gi/(float)(NS_Size)-0.5)+
                    8.0*(gj/(float)(WE_Size)-0.5)*(gj/(float)(WE_Size)-0.5);
            }
        }
    }
    else
    {
        for( gi=0;  gi < offset; gi++)
        {
            for( gj=1; gj<=WE_Size-2; gj++)
            {
                pos=getPos(gi,gj);
                Rabbit[pos] =
                    128.0*(gi-1)*(NS_Size-gi)*(gj-1)*(WE_Size-gj) /
                    (float)(NS_Size*NS_Size*WE_Size*WE_Size);
                Fox[pos] =
                    8.0*(gi/(float)(NS_Size)-0.5)*(gi/(float)(NS_Size)-0.5)+
                    8.0*(gj/(float)(WE_Size)-0.5)*(gj/(float)(WE_Size)-0.5);
            }
        }
    }
    return(err);
}


int nonCriticalLines(float* Rabbit,float* Fox,float* TRabbit,float* TFox,int offset)
{
    int gi,gj;
    int err = 0,pos;

    for( gi=1; gi < offset-1; gi++)
    {
        for( gj=1; gj<=WE_Size-3; gj++)
```

```
        {
            pos=getPos(gi,gj);
            TRabbit[pos] = (1.0+AlR-4.0*MuR)*Rabbit[pos] +
                           BtR*Fox[pos] +
                           MuR*(Rabbit[pos-1]+Rabbit[pos+1]+Rabbit[getPos(gi-1,gj)]+Rabbi

            TFox[pos] = AlF*Rabbit[pos] +
                        (1.0+BtF-4.0*MuF)*Fox[pos] +
                        MuF*(Fox[pos-1]+Fox[pos+1]+Fox[getPos(gi-1,gj)]+Fox[getPos(gi+1,g
        }
    }

    for( gi=1; gi < offset-1; gi++)
    {
        for(gj=1; gj<=WE_Size-3; gj++ )
        {
            pos=getPos(gi,gj);
            TRabbit[pos] = MAX( 0.0, TRabbit[pos]);
            TFox[pos] = MAX( 0.0, TFox[pos]);
        }
    }

    return (err);
}

int criticalLines(float* lineUpR,float* lineDownR,float* lineUpF,float* lineDownF,
    float* Rabbit,float* Fox,float* TRabbit,float* TFox, int my_rank, int comm_sz,int offs
{
    int gj,pos;
    int err = 0;
    if(my_rank==0)
    {

        for( gj=1; gj<=WE_Size-2; gj++)
        {
            pos=getPos(offset-1,gj);
            TRabbit[pos] = (1.0+AlR-4.0*MuR) * Rabbit[pos] +
                                    BtR * Fox[pos] +
                                    MuR*(Rabbit[pos-1] + Rabbit[pos+1] +
                                    Rabbit[getPos(offset-2,gj)] + lineDownR[gj]);
            TFox[pos] = AlF * Rabbit[pos] +
                                (1.0+BtF-4.0*MuF) * Fox[pos] +
                                MuF * (Fox[pos-1] + Fox[pos+1] +
                                Fox[getPos(offset-2,gj)] + lineDownF[gj]);
            TRabbit[pos] = MAX( 0.0, TRabbit[pos]);
            TFox[pos] = MAX( 0.0, TFox[pos]);

        }
    }
```

```
if ((my_rank + 1) == comm_sz)
{
    for( gj=1; gj<=WE_Size-2; gj++)
    {
        pos=getPos(0,gj);
        TRabbit[pos] = (1.0+AlR-4.0*MuR)*Rabbit[pos] +
                                BtR*Fox[pos] +
                                MuR*(Rabbit[pos-1]+Rabbit[pos+1]+
                                lineUpR[gj]+Rabbit[getPos(1,gj)]);

        TFox[pos] = AlF*Rabbit[pos] +
                                (1.0+BtF-4.0*MuF)*Fox[pos] +
                                MuF*(Fox[pos-1]+Fox[pos+1]+
                                lineUpF[gj]+Fox[getPos(1,gj)]);
        TRabbit[pos] = MAX( 0.0, TRabbit[pos]);
        TFox[pos] = MAX( 0.0, TFox[pos]);

    }
}
else
{
    for( gj=1; gj<=WE_Size-2; gj++)
    {
        pos=getPos(0,gj);
        TRabbit[pos] = (1.0+AlR-4.0*MuR)*Rabbit[pos] +
                                BtR*Fox[pos] +
                                MuR*(Rabbit[pos-1]+Rabbit[pos+1]+
                                lineUpR[gj]+Rabbit[getPos(1,gj)]);

        TFox[pos] = AlF*Rabbit[pos] +
                            (1.0+BtF-4.0*MuF)*Fox[pos] +
                            MuF*(Fox[pos-1]+Fox[pos+1]+
                            lineUpF[gj]+Fox[getPos(1,gj)]);

        TRabbit[pos] = MAX( 0.0, TRabbit[pos]);
        TFox[pos] = MAX( 0.0, TFox[pos]);
    }
    for( gj=1; gj<=WE_Size-2; gj++)
    {
        pos=getPos(offset-1,gj);
        TRabbit[pos] = (1.0+AlR-4.0*MuR)*Rabbit[pos] + BtR*Fox[pos] +
                            MuR*(Rabbit[pos-1] + Rabbit[pos+1]
                            + Rabbit[getPos(offset-2,gj)]+lineDownR[gj]);

        TFox[pos] = AlF*Rabbit[pos] +
                            (1.0+BtF-4.0*MuF)*Fox[pos] +
                            MuF*(Fox[pos-1]+Fox[pos+1]+
                            Fox[getPos(offset-2,gj)]+lineDownF[gj]);
        TRabbit[pos] = MAX( 0.0, TRabbit[pos]);
        TFox[pos] = MAX( 0.0, TFox[pos]);
```

```
        }
    }
    return(err);
}


int FillBorder(float  *Animal,int line_offset, float* lineUp, float* lineDown, int rank, i
{
    int        err;
    int        i, j;
    err = 0;

    if(rank == 0)
    {

        for( j=0; j<=WE_Size-2; j++)
        {
            Animal[getPos(0,j)] = lineUp[j];
        }
    }
    if (rank +1 == comm_size)
    {
        for( j=0; j<=WE_Size-2; j++)
        {
            Animal[getPos(line_offset-1,j)] = lineDown[j];
        }
    }
    for( i=0; i < line_offset; i++)
    {
        Animal[getPos(i,0)] = Animal[getPos(i,WE_Size-2)];
        Animal[getPos(i,WE_Size-1)] = Animal[getPos(i,1)];
    }
    return(err);
}


int GetPopulation(float *Animal,float *tcount,int offset,int rank,int comm_size)
{
    int   err;
    int   i, j;
    float p;

    err = 0;
    p = 0.0;
    if(rank == 0)
    {
        for( i=1; i < offset; i++)
            for( j=1;j<=WE_Size-2; j++)
                p = p + Animal[getPos(i,j)];
    }
```

```
    if(rank == comm_size-1)
    {
        for( i=0; i < offset -1; i++)
            for( j=1;j<=WE_Size-2; j++)
                p = p + Animal[getPos(i,j)];
    }
    else
    {
        for( i=0; i < offset; i++)
            for( j=1;j<=WE_Size-2; j++)
                p = p + Animal[getPos(i,j)];
    }
    *tcount = p;
    return(err);
}
```