###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Игра "Жизнь" Дж. Конвея»

студента 2 курса, 20204 группы

**Дронова Дениса Юрьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc103384010)

[ЗАДАНИЕ 3](#_Toc103384011)

[ХОД РАБОТЫ 4](#_Toc103384012)

[ЗАКЛЮЧЕНИЕ 5](#_Toc103384013)

[Приложение 1. *Таблица и графики* 6](#_Toc103384014)

[Приложение 2. *Профилирование* 8](#_Toc103384015)

[Приложение 3. *Код программы* 9](#_Toc103384016)

# ЦЕЛЬ

Практическое освоение методов реализации алгоритмов мелкозернистого параллелизма на крупноблочном параллельном вычислительном устройстве на примере реализации клеточного автомата «Игра "Жизнь" Дж. Конвея» с использованием неблокирующих коммуникаций библиотеки MPI.

# ЗАДАНИЕ

1. Написать параллельную программу на языке C/C++ с использованием MPI, реализующую клеточный автомат игры "Жизнь" с завершением программы по повтору состояния клеточного массива в случае одномерной декомпозиции массива по строкам и с циклическими границами массива. Проверить корректность исполнения алгоритма на различном числе процессорных ядер и различных размерах клеточного массива, сравнив с результатами, полученными для исходных данных вручную.
2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16, …. Размеры клеточного массива X и Y подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы, ускорения и эффективности распараллеливания от числа используемых ядер.
3. Произвести профилирование программы и выполнить ее оптимизацию. Попытаться достичь 50-процентной эффективности параллельной реализации на 16 ядрах для выбранных X и Y.

# ХОД РАБОТЫ

1. Была написана программа на C++ с использованием MPI, реализующая клеточный автомат игры "Жизнь" с завершением программы по повтору состояния клеточного массива в случае одномерной декомпозиции массива по строкам и с циклическими границами массива. Проверена корректность исполнения алгоритма на различном числе процессорных ядер и различных размерах клеточного массива.
2. Были подобраны размеры массива так, что решение задачи на одном ядре заняло не менее 30 секунд (X = 250, Y = 250). Измерено время работы программы на различном числе ядер: 1, 2, 4, 8, 12, 16, 24. На основе измерений были построены таблицы и графики времени исполнения программы, эффективности и ускорения распараллеливания в зависимости от числа используемых ядер (см. Приложение).
3. Выполнено профилирование программы с использованием 16 ядер. Результаты приведены в приложении

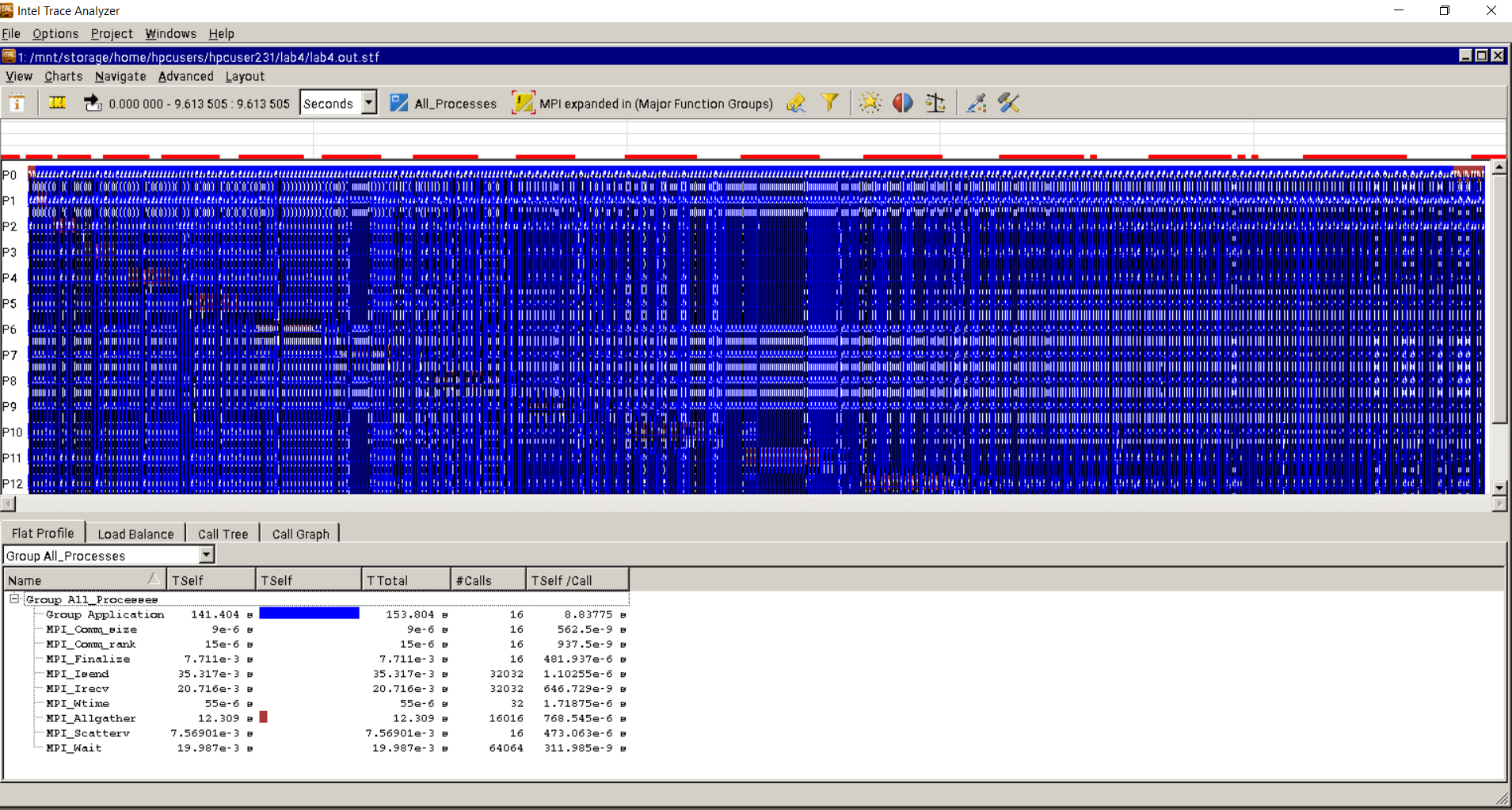
# ЗАКЛЮЧЕНИЕ

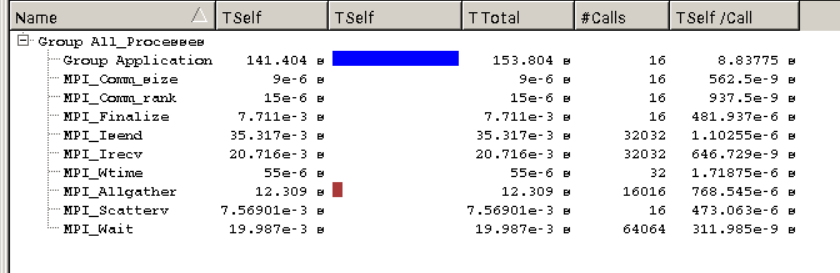
Можно сделать вывод о том, что распараллеливание игры “Жизнь” является целесообразным, так как на основе данных таблиц и графиков наблюдается ускорение программы. Однако с увеличением числа процессов/ядер увеличиваются накладные расходы на их общение, что может привести к увеличению времени исполнения программ и снижению эффективности. Также после анализа профилирования программы становится понятно, что низкая эффективность наблюдается в силу неоднородности распределения данных между процессорами на примере глайдера.

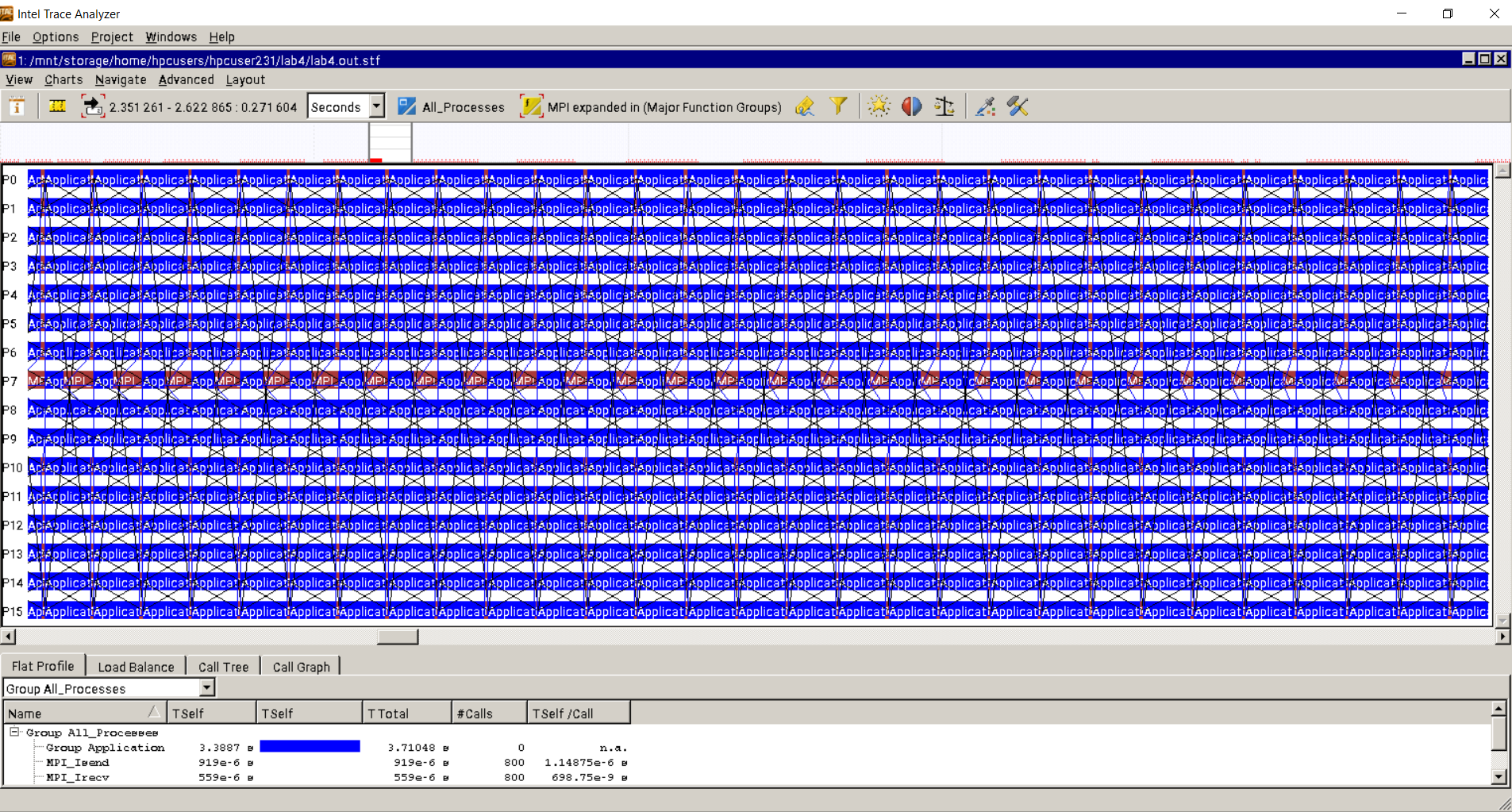
# Приложение 1. *Таблица и графики*

|  |  |  |  |
| --- | --- | --- | --- |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 50,06700 | 100% | 1 |
| 2 | 56,65190 | 44% | 0,883765593 |
| 4 | 35,27870 | 35% | 1,419184947 |
| 8 | 18,70520 | 33% | 2,676635374 |
| 12 | 12,62770 | 33% | 3,964855041 |
| 16 | 9,63322 | 32% | 5,197327581 |
| 24 | 6,71577 | 31% | 7,455139172 |

# Приложение 2. *Профилирование*







# Приложение 3. *Код программы*

#include <iostream>

#include "mpi.h"

#define WIDTH 250

#define HEIGHT 250

#define DEAD false

#define ALIVE true

void initField(bool\* field, int height, int width)

{

    for (int i = 0; i < height \* width; ++i)

    {

        field[i] = DEAD;

    }

    field[2 \* WIDTH + 0] = ALIVE;

    field[2 \* WIDTH + 1] = ALIVE;

    field[2 \* WIDTH + 2] = ALIVE;

    field[1 \* WIDTH + 2] = ALIVE;

    field[0 \* WIDTH + 1] = ALIVE;

}

void calculateStopFlags(bool\* fieldPart, bool\*\* history, int iterCounter, int width, int height, bool\* local\_similarity)

{

    for (int i = 0; i < iterCounter; i++)

    {

        local\_similarity[i] = false;

    }

    int simiarityCount = 0;

    bool exitFlag = false;

    for (int i = 0; i < iterCounter; i++)

    {

        bool\* part = history[i]; // itearating through field parts history

        for (int j = 0; j < height; j++)

        {

            for (int k = 0; k < width; k++)

            {

                if (part[j \* width + k] == fieldPart[j \* width + k])

                {

                    simiarityCount++;

                }

                else

                {

                    exitFlag = true;

                    break;

                }

            }

            if (exitFlag)

            {

                exitFlag = false;

                break;

            }

        }

        if (simiarityCount == width \* height) // cuurent field state repeats one of the previous ones

        {

            local\_similarity[i] = true;

        }

        else

        {

            local\_similarity[i] = false;

            simiarityCount = 0;

        }

    }

}

void simualteEra(bool\* part, int width, int height, bool\* firstRow, bool\* penultRow)

{

    int\* aliveAround = new int[width \* height];

    for (int i = 0; i < width \* height; ++i)

    {

        aliveAround[i] = 0;

    }

    for (int i = 0; i < width; i++)

    {

        firstRow[i] = part[width + i];                 // first row (zero row is before)

        penultRow[i] = part[width \* (height - 2) + i]; // penultimate row

    }

    for (int i = 1; i < height - 1; i++)

    {

        for (int j = 0; j < width; j++)

        {

            int idx = i \* width + j;

            if (j == (width - 1))

            {

                aliveAround[idx] += int(part[idx - 1]) + int(part[idx + 1])

                    + int(part[idx + width]) + int(part[idx - width])

                    + int(part[idx + width - 1]) + int(part[idx - width - 1])

                    + int(part[idx + 1 - width]) + int(part[idx + 1 - 2 \* width]);

            }

            else if (j == 0)

            {

                aliveAround[idx] += int(part[idx - 1]) + int(part[idx + 1])

                    + int(part[idx + width]) + int(part[idx - width])

                    + int(part[idx + 1 - width]) + int(part[idx + 1 + width])

                    + int(part[idx + 2 \* width - 1]) + int(part[idx - 1 + width]);

            }

            else

            {

                aliveAround[idx] += int(part[idx - 1]) + int(part[idx + 1])

                    + int(part[idx + 1 + width]) + int(part[idx - 1 + width])

                    + int(part[idx + width]) + int(part[idx - width])

                    + int(part[idx + 1 - width]) + int(part[idx - 1 - width]);

            }

        }

    }

    for (int i = 1; i < height - 1; i++)

    {

        for (int j = 0; j < width; j++)

        {

            int idx = i \* width + j;

            if (part[idx] == ALIVE)

            {

                if (aliveAround[idx] > 3 || aliveAround[idx] < 2)

                {

                    part[idx] = DEAD;

                }

                else {

                    part[idx] = ALIVE;

                }

            }

            else

            {

                if (aliveAround[idx] == 3)

                {

                    part[idx] = ALIVE;

                }

            }

        }

    }

    delete[] aliveAround;

}

void simulateRemainingCells(bool\* part, bool\* topLine, bool\* bottomLine, int width, int displ)

{

    int\* aliveAround = new int[width];

    for (int i = 0; i < width; ++i)

    {

        aliveAround[i] = 0;

    }

    for (int i = 0; i < width; i++)

    {

        if (i == 0)

        {

            aliveAround[i] += int(topLine[i]) + int(topLine[i + 1])

                + int(topLine[width - 1]) + int(bottomLine[i])

                + int(bottomLine[i + 1]) + int(bottomLine[width - 1])

                + int(part[i + 1 + displ]) + int(part[width - 1 + displ]);

        }

        else if (i == width - 1)

        {

            aliveAround[i] += int(topLine[i]) + int(topLine[i - 1])

                + int(topLine[0]) + int(bottomLine[i])

                + int(bottomLine[i - 1]) + int(bottomLine[0])

                + int(part[i - 1 + displ]) + int(part[0 + displ]);

        }

        else

        {

            aliveAround[i] += int(topLine[i]) + int(topLine[i - 1])

                + int(topLine[i + 1]) + int(bottomLine[i])

                + int(bottomLine[i - 1]) + int(bottomLine[i + 1])

                + int(part[i - 1 + displ]) + int(part[i + 1 + displ]);

        }

    }

    for (int i = 0; i < width; i++)

    {

        if (part[i + displ] == ALIVE)

        {

            if (aliveAround[i] > 3 || aliveAround[i] < 2)

            {

                part[i + displ] = DEAD;

            }

            else {

                part[i + displ] = ALIVE;

            }

        }

        else

        {

            if (aliveAround[i] == 3)

            {

                part[i + displ] = ALIVE;

            }

        }

    }

    delete[] aliveAround;

}

bool allPartsRepeat(bool\* general\_similarity, int procSize, int iterCounter, int iterations)

{

    for (int i = 0; i < iterCounter; i++)

    {

        int counter = 0;

        for (int j = 0; j < procSize; j++)

        {

            if (general\_similarity[j \* iterations + i] == true)

            {

                counter++;

            }

            else

            {

                break;

            }

        }

        if (counter == procSize) // every process have duplicate part on a certain iteration

        {

            return true;

        }

        else

        {

            counter = 0;

        }

    }

    return false;

}

int startSimulation(int procSize, int procRank)

{

    int\* rowsNum = new int[procSize];

    int\* sendcounts = new int[procSize];

    int\* displs = new int[procSize];

    int rowsDefault = (HEIGHT / procSize);

    int dif = HEIGHT - (rowsDefault \* procSize);

    int displ = 0;

    for (int i = 0; i < procSize; i++)

    {

        rowsNum[i] = rowsDefault;

        if (i < dif)

        {

            rowsNum[i] += 1;

        }

        sendcounts[i] = rowsNum[i] \* WIDTH;

        displs[i] = displ;

        displ += sendcounts[i];

    }

    bool\* field = NULL;

    bool\* fieldPart = new bool[rowsNum[procRank] \* WIDTH];

    bool\* topLine = new bool[WIDTH];

    bool\* bottomLine = new bool[WIDTH];

    bool\* firstRow = new bool[WIDTH];

    bool\* penultRow = new bool[WIDTH];

    // array that stores history of field parts states for each iteration

    int itersRestrictor = 2000;

    bool\*\* history = new bool\* [itersRestrictor];

    for (int i = 0; i < itersRestrictor; i++)

    {

        history[i] = new bool[rowsNum[procRank] \* WIDTH];

    }

    bool\* general\_similarity = new bool[procSize \* itersRestrictor];

    bool\* local\_similarity = new bool[itersRestrictor];

    for (int i = 0; i < itersRestrictor; i++)

    {

        local\_similarity[i] = false;

    }

    if (procRank == 0)

    {

        field = new bool[HEIGHT \* WIDTH];

        initField(field, HEIGHT, WIDTH);

    }

    MPI\_Scatterv(field, sendcounts, displs, MPI\_C\_BOOL, fieldPart, sendcounts[procRank], MPI\_C\_BOOL, 0, MPI\_COMM\_WORLD);

    int next\_rank = procRank + 1;

    int prev\_rank = procRank - 1;

    if (next\_rank == procSize)

    {

        next\_rank = 0;

    }

    if (prev\_rank == -1)

    {

        prev\_rank = procSize - 1;

    }

    int iterCounter = 0;

    while (iterCounter < itersRestrictor)

    {

        MPI\_Request sendRequestUp;

        MPI\_Request sendRequestDown;

        MPI\_Request recvRequestUp;

        MPI\_Request recvRequestDown;

        MPI\_Request recvRequsetVectors;

        MPI\_Isend(fieldPart, WIDTH, MPI\_C\_BOOL, prev\_rank, 1, MPI\_COMM\_WORLD, &sendRequestUp);

        MPI\_Isend(fieldPart + WIDTH \* (rowsNum[procRank] - 1), WIDTH, MPI\_C\_BOOL, next\_rank, 0, MPI\_COMM\_WORLD, &sendRequestDown);

        MPI\_Irecv(topLine, WIDTH, MPI\_C\_BOOL, prev\_rank, 0, MPI\_COMM\_WORLD, &recvRequestUp);      // receiving upper line from previous process

        MPI\_Irecv(bottomLine, WIDTH, MPI\_C\_BOOL, next\_rank, 1, MPI\_COMM\_WORLD, &recvRequestDown); // receiving down line from next process

        memcpy(history[iterCounter], fieldPart, sendcounts[procRank]);

        calculateStopFlags(fieldPart, history, iterCounter, WIDTH, rowsNum[procRank], local\_similarity);

        MPI\_Iallgather(local\_similarity, itersRestrictor, MPI\_C\_BOOL, general\_similarity, itersRestrictor, MPI\_C\_BOOL, MPI\_COMM\_WORLD, &recvRequsetVectors);

        simualteEra(fieldPart, WIDTH, rowsNum[procRank], firstRow, penultRow);

        // Simulating remainig cells of field part using lines recieved from top and bottom processes

        MPI\_Wait(&sendRequestUp, MPI\_STATUS\_IGNORE);

        MPI\_Wait(&recvRequestUp, MPI\_STATUS\_IGNORE);

        simulateRemainingCells(fieldPart, topLine, firstRow, WIDTH, 0);

        MPI\_Wait(&sendRequestDown, MPI\_STATUS\_IGNORE);

        MPI\_Wait(&recvRequestDown, MPI\_STATUS\_IGNORE);

        simulateRemainingCells(fieldPart, penultRow, bottomLine, WIDTH, WIDTH \* (rowsNum[procRank] - 1));

        MPI\_Wait(&recvRequsetVectors, MPI\_STATUS\_IGNORE);

        if (allPartsRepeat(general\_similarity, procSize, iterCounter, itersRestrictor))

        {

            break;

        }

        iterCounter++;

    }

    delete[] bottomLine;

    delete[] fieldPart;

    delete[] topLine;

    delete[] local\_similarity;

    delete[] general\_similarity;

    delete[] firstRow;

    delete[] penultRow;

    for (int i = 0; i < itersRestrictor; i++)

    {

        delete[] history[i];

    }

    delete[] history;

    return iterCounter;

}

int main(int argc, char\*\* argv)

{

    int procSize;

    int procRank;

    double start, finish;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &procSize);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procRank);

    start = MPI\_Wtime();

    int iterationsNum = startSimulation(procSize, procRank);

    finish = MPI\_Wtime();

    if (procRank == 0)

    {

        std::cout << "Total time: " << finish - start << " sec." << std::endl;

        std::cout << "Iterations count: " << iterationsNum << std::endl;

    }

    MPI\_Finalize();

    return 0;

}