**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 ПРАКТИЧЕСКОЙ РАБОТЫ**

«Игра «Жизнь» Дж. Конвэя»

студента 2 курса, группы 21206

**Балашова Вячеслава Вадимовича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2023

**Содержание**

Цель3

Задание3

Описание работы4

Заключение10

Приложение 1. Исходный код программы11

**Цель**

1. 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.

**Описание работы**

Ход выполнения работы:

1. Была написана параллельная программа на языке Си с использованием библиотеки MPI, в которой для удобства все поле инициализируется на процессе с нулевым рангом, после чего части поля раздается всем процессам. После выполнения раздачи все процессы начинают выполнять итерации в цикле, пока состояние поля на одной из следующих итераций не совпадет с состоянием поля на одной из предыдущих итераций, либо пока количество итераций не достигнет максимального возможного числа (см. Приложение 1).
2. Для проведения замеров использовалось квадратное поле n\*n, где n = 500. В качестве стартовой конфигурации поля использовалась фигура «Глайдер», которая смещается на 1 клетку вправо и на 1 клетку вниз за 4 итерации. Количество итераций, после которых программа завершится, равно n \* 4 + 1.
3. Были проведены замеры времени работы программы на 1, 2, 4, 8 и 16 процессах с данными, описанных в пункте 2. Также были построены графики зависимости времени работы, ускорения и эффективности распараллеливания в зависимости от количества процессов.

Таблица 1. Соответствие времени работы программы и количества процессов для параллельной программы.

|  |  |
| --- | --- |
| Время, сек. | Количество процессов, шт. |
| 110,4729434 | 1 |
| 124,3876666 | 2 |
| 78,2641686 | 4 |
| 40,9602074 | 8 |
| 21,3435538 | 16 |

Таблица 2. Соответствие коэффициентов ускорения и количества процессов для параллельной программы.

|  |  |
| --- | --- |
| Ускорение, коэфф. | Количество процессов, шт. |
| 1 | 1 |
| 0,8881342212 | 2 |
| 1,411539219 | 4 |
| 2,697079688 | 8 |
| 5,17593951 | 16 |

Таблица 3. Соответствие эффективности и количества процессов параллельной программы.

|  |  |
| --- | --- |
| Эффективность, % | Количество процессов, шт. |
| 100,00% | 1 |
| 44,41% | 2 |
| 35,29% | 4 |
| 33,71% | 8 |
| 32,35% | 16 |

1. Было выполнено профилирование параллельной программы, запущенной на 24 процессах с помощью инструмента ITAC (Intel Trace Analyzer and Collector).

Рисунок 1. Профилирование параллельной программы: общий вид.

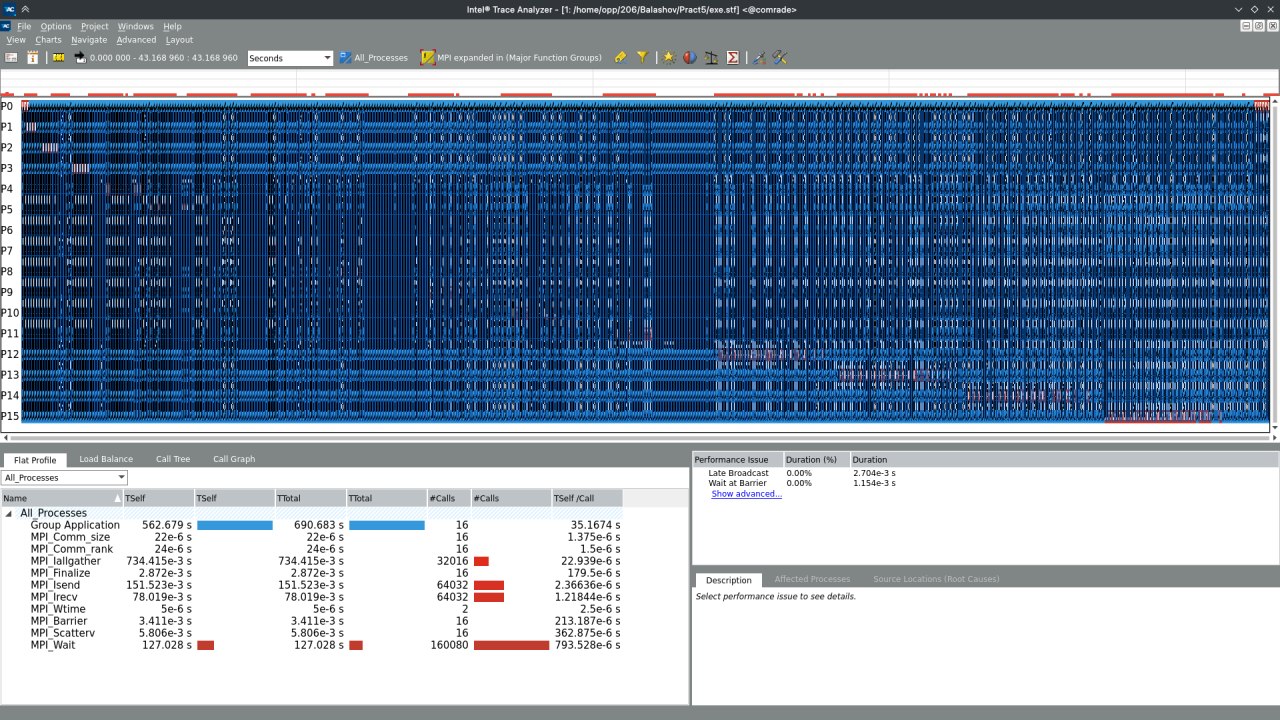


Рисунок 2. Профилирование параллельной программы: начало программы.

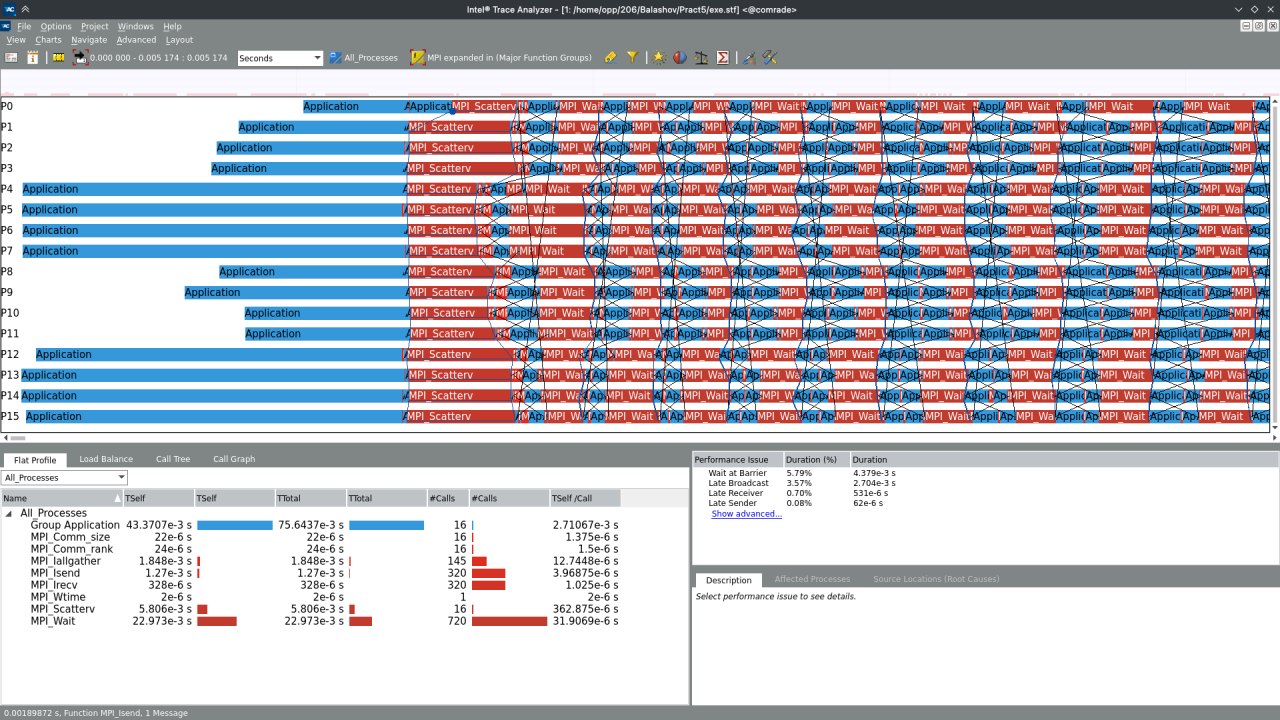


Рисунок 3. Профилирование параллельной программы: Приближенное рассмотрение.

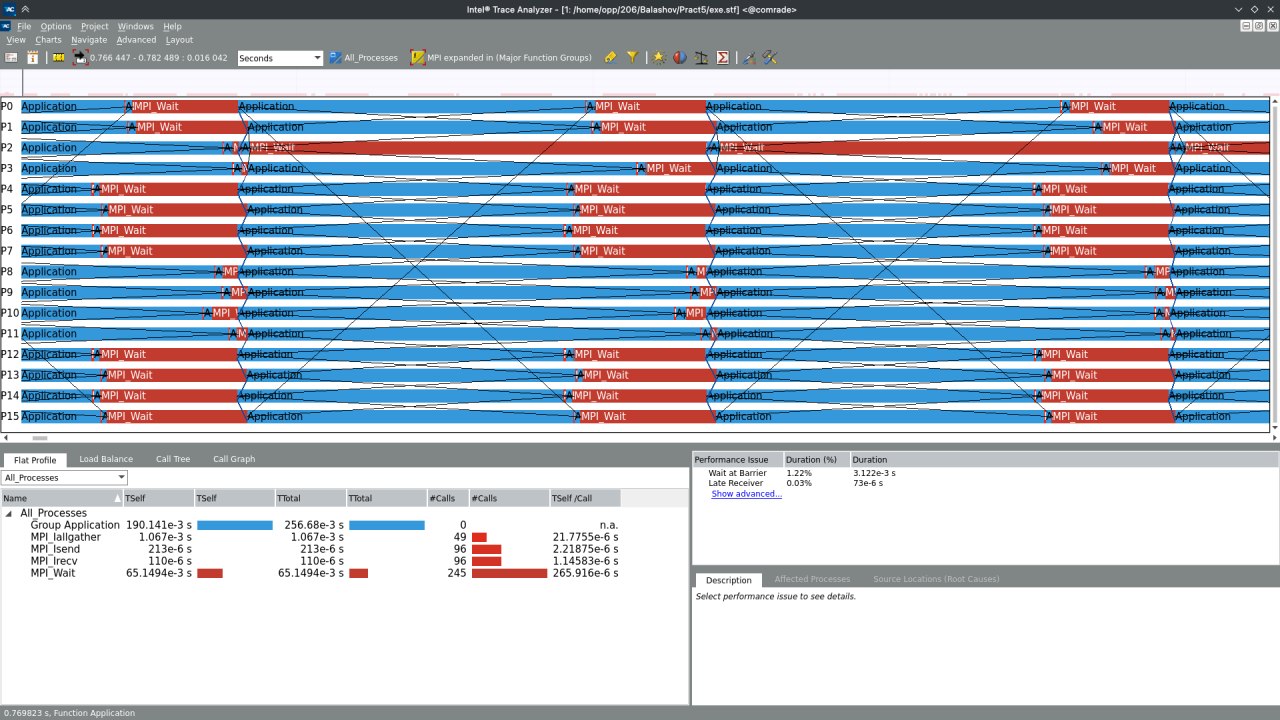


Рисунок 4. Профилирование параллельной программы: еще более приближенное рассмотрение.

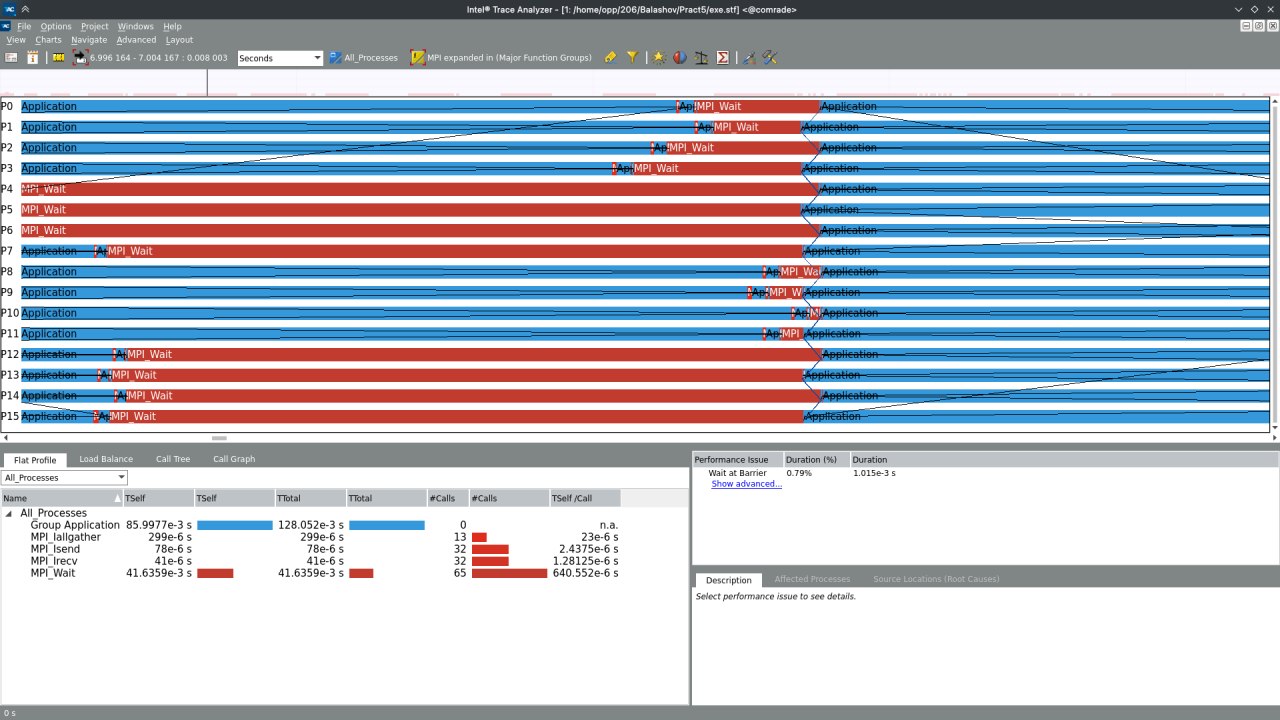


Рисунок 5. Профилирование параллельной программы: более близкий рассмотр Isend и Irecv.

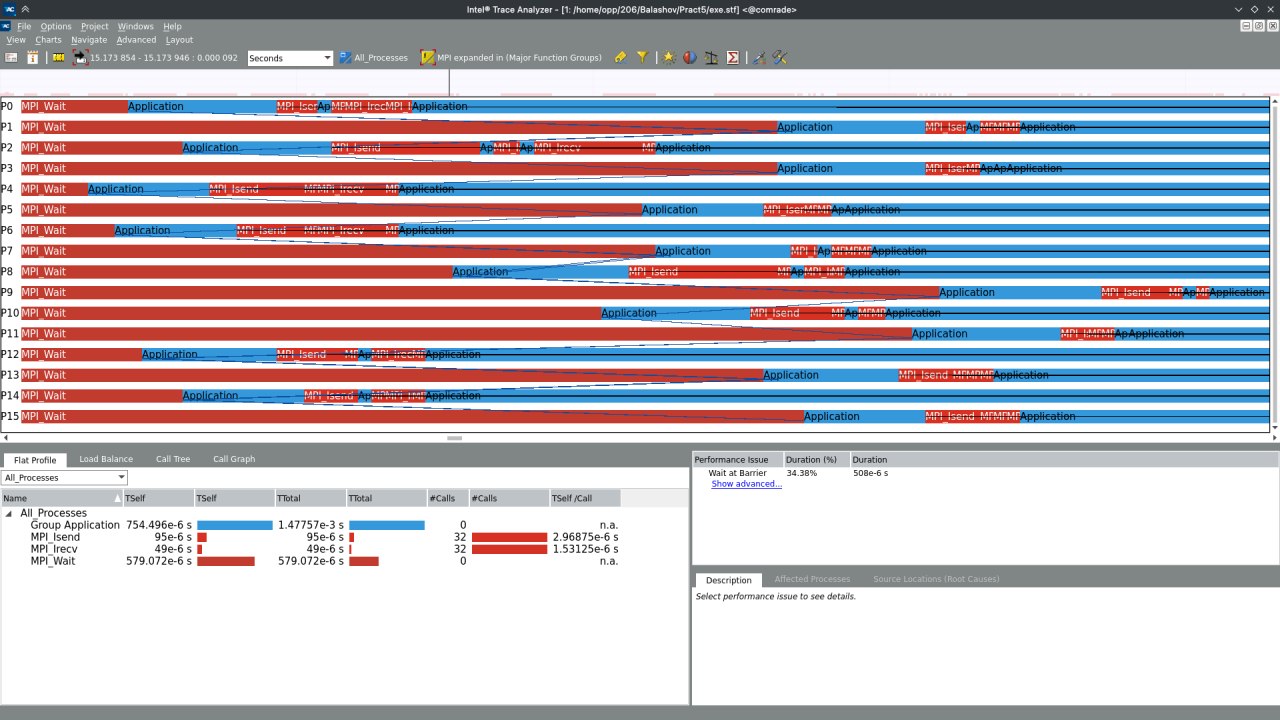
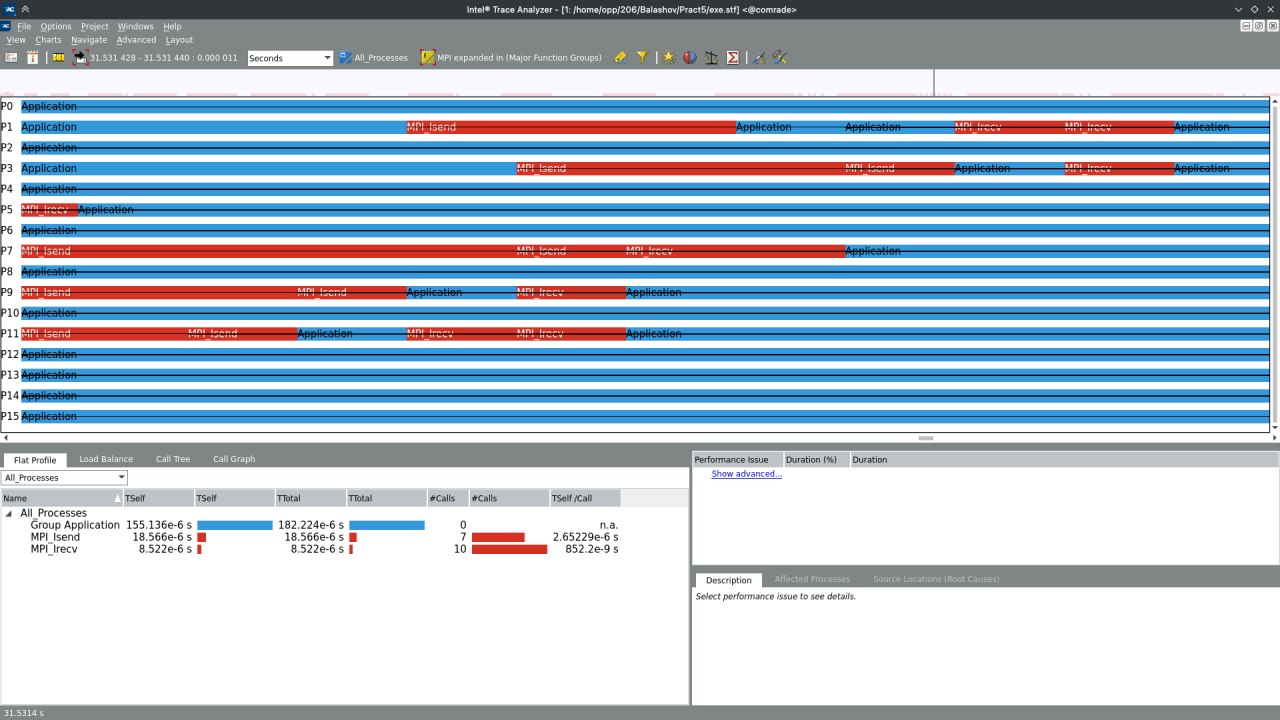


Рисунок 6. Профилирование параллельной программы: более близкий рассмотр Isend и Irecv.



**Заключение**

В ходе выполнения практической работы был получен опыт работы с неблокирующими функциями библиотеки MPI. Особенность этих функций заключается в том, что передача данных между процессами происходит параллельно с самой программой, что можно заметить при профилировании программы, ведь вызов этих функций (или же инициализация передачи данных) занимает мало времени.

**Приложение 1. Исходный код программы.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <mpi.h>

void initFieldWithGliderTopLeftCorner(char \* field, int height, int width)

{

memset(field, 0, width \* height);

field[0 \* width + 1] = 1;

field[1 \* width + 2] = 1;

field[2 \* width + 0] = 1;

field[2 \* width + 1] = 1;

field[2 \* width + 2] = 1;

}

void printField(char \* field, int height, int width)

{

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

{

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

{

printf("%d ", field[i \* width + j]);

}

printf("\n");

}

}

char countNewStageBySum(char cellState, int sum)

{

if (cellState)

{

if (sum < 2 || sum > 3)

{

return 0;

}

return 1;

}

else

{

if (sum == 3)

{

return 1;

}

return 0;

}

}

char countNewStageByNeighbours(char cellState, char one, char two, char three, char four, char five, char six, char seven, char eight)

{

int sum = one + two + three + four + five + six + seven + eight;

return countNewStageBySum(cellState, sum);

}

char countNewStageByIndex(char \* field, int index, int fieldWidth)

{

int sum = field[index - 1] + field[index + 1] + field[index - fieldWidth - 1] + field[index - fieldWidth] + field[index - fieldWidth + 1] +

field[index + fieldWidth - 1] + field[index + fieldWidth] + field[index + fieldWidth + 1];

return countNewStageBySum(field[index], sum);

}

void updateFieldLine(char \* fieldLineSrc, char \* fieldLineDst, int fieldWidth)

{

\*fieldLineDst = countNewStageByNeighbours(\*fieldLineSrc,

\*(fieldLineSrc - fieldWidth + fieldWidth - 1),

\*(fieldLineSrc - fieldWidth),

\*(fieldLineSrc - fieldWidth + 1),

\*(fieldLineSrc + fieldWidth - 1),

\*(fieldLineSrc + 1),

\*(fieldLineSrc + fieldWidth + fieldWidth - 1),

\*(fieldLineSrc + fieldWidth),

\*(fieldLineSrc + fieldWidth + 1));

for (int k = 1; k < fieldWidth - 1; ++k)

{

\*(fieldLineDst + k) = countNewStageByIndex(fieldLineSrc, k, fieldWidth);

}

\*(fieldLineDst + fieldWidth - 1) = countNewStageByNeighbours(\*(fieldLineSrc + fieldWidth - 1),

\*(fieldLineSrc + fieldWidth - 1 - fieldWidth - 1),

\*(fieldLineSrc + fieldWidth - 1 - fieldWidth),

\*(fieldLineSrc + fieldWidth - 1 - fieldWidth - fieldWidth + 1),

\*(fieldLineSrc + fieldWidth - 1 - 1),

\*(fieldLineSrc + fieldWidth - 1 - fieldWidth + 1),

\*(fieldLineSrc + fieldWidth - 1 + fieldWidth - 1),

\*(fieldLineSrc + fieldWidth - 1 + fieldWidth),

\*(fieldLineSrc + fieldWidth - 1 + 1));

}

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

{

int fieldHeight = 100;

int fieldWidth = 100;

int maxNumOfIterations = 100;

if (argc > 1)

{

fieldHeight = atoi(argv[1]);

}

if (argc > 2)

{

fieldWidth = atoi(argv[2]);

}

if (argc > 3)

{

maxNumOfIterations = atoi(argv[3]);

}

const int ROOT\_RANK = 0;

const int PREV\_TO\_NEXT\_RANK\_MESSAGE\_ID = 101;

const int NEXT\_TO\_PREV\_RANK\_MESSAGE\_ID = 102;

int mpiSize;

int mpiRank;

int mpiNextRank;

int mpiPrevRank;

char \* field;

int \* sendCounts;

int \* displs;

double start;

char \* fieldPart;

char \* fieldPartBuf;

char \* stopFlags;

char \*\* previousStages;

int currSegmentHeight;

int minSegmentHeight;

int heightRemains;

MPI\_Request sendToPrevReq;

MPI\_Request sendToNextReq;

MPI\_Request recvFromPrevReq;

MPI\_Request recvFromNextReq;

MPI\_Request gatherStopFlagsReq;

MPI\_Status mpiStatus;

MPI\_Init(&argc, &argv);

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

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

minSegmentHeight = fieldHeight / mpiSize;

heightRemains = fieldHeight - minSegmentHeight \* mpiSize;

currSegmentHeight = minSegmentHeight + ((mpiRank < heightRemains) ? 1 : 0);

if (mpiRank == ROOT\_RANK)

{

printf("Field height = %d\nField width = %d\nNumber Of iterations = %d\n", fieldHeight, fieldWidth, maxNumOfIterations);

start = MPI\_Wtime();

field = (char \*) malloc(sizeof(char) \* fieldWidth \* fieldHeight);

initFieldWithGliderTopLeftCorner(field, fieldHeight, fieldWidth);

sendCounts = (int \*) malloc(sizeof(int) \* mpiSize);

displs = (int \*) malloc(sizeof(int) \* mpiSize);

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

{

sendCounts[i] = (minSegmentHeight + ((i < heightRemains) ? 1 : 0)) \* fieldWidth;

displs[i] = (minSegmentHeight \* i + ((i < heightRemains) ? i : heightRemains)) \* fieldWidth;

}

}

fieldPart = (char \*) malloc(sizeof(char) \* (currSegmentHeight + 2) \* fieldWidth);

stopFlags = (char \*) malloc(sizeof(char) \* maxNumOfIterations \* mpiSize);

previousStages = (char \*\*) malloc(sizeof(char \*) \* maxNumOfIterations);

MPI\_Scatterv(field, sendCounts, displs, MPI\_CHAR, fieldPart + fieldWidth, currSegmentHeight \* fieldWidth, MPI\_CHAR, ROOT\_RANK, MPI\_COMM\_WORLD);

mpiPrevRank = mpiRank ? mpiRank - 1 : mpiSize - 1;

mpiNextRank = mpiRank < mpiSize - 1 ? mpiRank + 1 : 0;

int i = 0;

for (; i < maxNumOfIterations; ++i)

{

MPI\_Isend(fieldPart + fieldWidth, fieldWidth, MPI\_CHAR, mpiPrevRank, NEXT\_TO\_PREV\_RANK\_MESSAGE\_ID, MPI\_COMM\_WORLD, &sendToPrevReq);

MPI\_Isend(fieldPart + (currSegmentHeight) \* fieldWidth, fieldWidth, MPI\_CHAR, mpiNextRank, PREV\_TO\_NEXT\_RANK\_MESSAGE\_ID, MPI\_COMM\_WORLD, &sendToNextReq);

MPI\_Irecv(fieldPart, fieldWidth, MPI\_CHAR, mpiPrevRank, PREV\_TO\_NEXT\_RANK\_MESSAGE\_ID, MPI\_COMM\_WORLD, &recvFromPrevReq);

MPI\_Irecv(fieldPart + (1 + currSegmentHeight) \* fieldWidth, fieldWidth, MPI\_CHAR, mpiNextRank, NEXT\_TO\_PREV\_RANK\_MESSAGE\_ID, MPI\_COMM\_WORLD, &recvFromNextReq);

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

{

stopFlags[mpiRank \* maxNumOfIterations + j] = 1;

for (int k = 0; k < currSegmentHeight \* fieldWidth; ++k)

{

if (previousStages[j][fieldWidth + k] != fieldPart[fieldWidth + k])

{

stopFlags[mpiRank \* maxNumOfIterations + j] = 0;

break;

}

}

}

MPI\_Iallgather(stopFlags + mpiRank \* maxNumOfIterations, maxNumOfIterations, MPI\_CHAR, stopFlags, maxNumOfIterations, MPI\_CHAR, MPI\_COMM\_WORLD, &gatherStopFlagsReq);

fieldPartBuf = (char \*) malloc(sizeof(char) \* (currSegmentHeight + 2) \* fieldWidth);

for (int k = 1; k < currSegmentHeight - 1; ++k)

{

updateFieldLine(fieldPart + fieldWidth \* (k + 1), fieldPartBuf + fieldWidth \* (k + 1), fieldWidth);

}

MPI\_Wait(&sendToPrevReq, &mpiStatus);

MPI\_Wait(&recvFromPrevReq, &mpiStatus);

updateFieldLine(fieldPart + fieldWidth, fieldPartBuf + fieldWidth, fieldWidth);

MPI\_Wait(&sendToNextReq, &mpiStatus);

MPI\_Wait(&recvFromNextReq, &mpiStatus);

updateFieldLine(fieldPart + (1 + currSegmentHeight - 1) \* fieldWidth,

fieldPartBuf + (1 + currSegmentHeight - 1) \* fieldWidth, fieldWidth);

MPI\_Wait(&gatherStopFlagsReq, &mpiStatus);

previousStages[i] = fieldPart;

fieldPart = fieldPartBuf;

char finish = 0;

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

{

finish = 1;

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

{

if (!stopFlags[k \* maxNumOfIterations + j])

{

finish = 0;

break;

}

}

if (finish)

{

break;

}

}

if (finish)

{

break;

}

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (mpiRank == ROOT\_RANK)

{

printf("Took %d iterations for %lf sec for trying to return to one of previous stages\n", i + 1, MPI\_Wtime() - start);

}

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

{

free(previousStages[j]);

}

if (i != maxNumOfIterations)

{

free(previousStages[i]);

}

free(previousStages);

free(stopFlags);

free(fieldPart);

if (mpiRank == ROOT\_RANK)

{

free(field);

free(displs);

free(sendCounts);

}

MPI\_Finalize();

return 0;

}