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, 21204 группы

Осипова Александра Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

ЦЕЛЬ

Практическое освоение функционала неблокирующих коммуникаций в MPI на примере реализации клеточного автомата «Игра "Жизнь" Дж. Конвея».

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

1. В качестве исходных данных было выбрано поле размером 400x400 с «глайдером». Если клетка «живая», то ее значение в массиве равно единице, иначе – нулю.

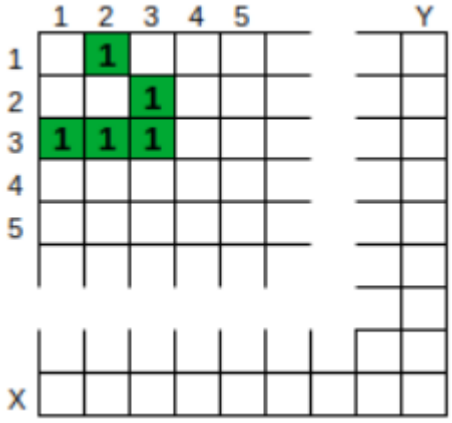


Рис.1 Исходное заполнение клеточного массива

2. Была написана последовательная программа (см. Приложение 1).
3. Была написана параллельная программа, в которой использовались неблокирующие MPI коммуникации (см. Приложение 2).

3.1 В ней исходное поле распределяется по блокам строк между процессами.

```
int *sizesOfParts = CalcSizesOfParts(numProc, numRows, numCols);
int *displsOfParts = CalcDisplsOfParts(numProc, sizesOfParts);

char *partOfField = (char *) malloc(sizesOfParts[rank] * sizeof(char));

MPI_Scatterv(field, sizesOfParts, displsOfParts, MPI_CHAR, partOfField,
sizesOfParts[rank], MPI_CHAR, 0, MPI_COMM_WORLD);
```

3.2 Далее каждый процесс, пока обменивается крайними строками со своими соседями, вычисляет следующее поколение для «внутренних» (не граничных) строк.

3.3 Вместе с этим он вычисляет булев вектор флага останова. Компонент этого вектора равен 1, если текущее состояние участка массива этого ядра совпадает с состоянием на соответствующей этому компоненту предыдущей итерации, и равен нулю – в противном случае. После того, как все ядра произведут обмен этими векторами флагов, каждое ядро сможет однозначно определить, нужно ли завершать программу. Останов происходит, когда один и тот же компонент вектора флагов равен 1 во всех присланных векторах.

3.4 Реализация функции, реализующий описанный выше алгоритм:

```
void StartSimulation(char* curPartOfField, int numRows, int numCols, int* sizesOfParts,
int numProc, int curcore, char* field, int* displs) {
     char** prevPartOfFields = (char**)malloc(NUM_ITER * sizeof(char*));

     char* stopFlags = (char*)calloc(NUM_ITER, sizeof(char));
     char* exchStopFlags = (char*)calloc(NUM_ITER, sizeof(char));

     int numRowsInCore = sizesOfParts[curcore] / numCols;

     int prevCore = (curcore == 0 ? numProc - 1 : curcore - 1);
     int nextCore = (curcore == numProc - 1 ? 0 : curcore + 1);

     char* lineFromPrevCore = (char*)malloc(numCols * sizeof(char));
     char* lineFromNextCore = (char*)malloc(numCols * sizeof(char));

     MPI_Request reqSendPrev;
     MPI_Request reqSendNext;
     MPI_Request reqRecvPrev;
     MPI_Request reqRecvNext;
     MPI_Request reqFlags;

     int countIter;

     for (countIter = 1; countIter <= NUM_ITER; ++countIter) {

           int numPassedIter = countIter - 1;

           MPI_Isend(curPartOfField, numCols, MPI_CHAR, prevCore, 0, MPI_COMM_WORLD,
&reqSendPrev);

           int shiftToLastLine = sizesOfParts[curcore] - numCols;
           MPI_Isend(curPartOfField + shiftToLastLine, numCols, MPI_CHAR, nextCore, 0,
MPI_COMM_WORLD, &reqSendNext);

           MPI_Irecv(lineFromPrevCore, numCols, MPI_CHAR, prevCore, MPI_ANY_TAG,
MPI_COMM_WORLD, &reqRecvPrev);

           MPI_Irecv(lineFromNextCore, numCols, MPI_CHAR, nextCore, MPI_ANY_TAG,
MPI_COMM_WORLD, &reqRecvNext);

           CalcVectorOfStopFlags(stopFlags, prevPartOfFields, curPartOfField,
sizesOfParts[curcore], countIter);

           int sizeOfIterBlock = ceil((double)numPassedIter / numProc);
           MPI_Ialltoall(stopFlags, sizeOfIterBlock, MPI_CHAR, exchStopFlags,
sizeOfIterBlock, MPI_CHAR, MPI_COMM_WORLD, &reqFlags);

           char* newPartOfField = (char*)malloc(sizesOfParts[curcore] * sizeof(char));

           CalcNextGenerationInCentre(newPartOfField, curPartOfField, numRowsInCore,
numCols);

           MPI_Wait(&reqSendPrev, MPI_STATUS_IGNORE);

           MPI_Wait(&reqRecvPrev, MPI_STATUS_IGNORE);

           CalcNextGenerationInRow(newPartOfField, curPartOfField, lineFromPrevCore, 0,
numRowsInCore, numCols, curcore);

           MPI_Wait(&reqSendNext, MPI_STATUS_IGNORE);

           MPI_Wait(&reqRecvNext, MPI_STATUS_IGNORE);
```

```
            CalcNextGenerationInRow(newPartOfField, curPartOfField, lineFromNextCore,
numRowsInCore - 1, numRowsInCore, numCols, curcore);

            MPI_Wait(&reqFlags, MPI_STATUS_IGNORE);

            if (IsMatchFound(exchStopFlags, sizeOfIterBlock, numPassedIter, numProc)) {
                  --countIter;
                  break;
            }
            prevPartOfFields[numPassedIter] = curPartOfField;
            curPartOfField = newPartOfField;
      }

      if (curcore == 0) {
            cout << "Total iterations: " << countIter << endl;
      }

      for (int i = 0; i < countIter; ++i) {
            free(prevPartOfFields[i]);
      }
      free(prevPartOfFields);
      free(stopFlags);
      free(exchStopFlags);
      free(lineFromNextCore);
      free(lineFromPrevCore);
}
```

4. Было измерено время работы последовательной программы.
5. Было измерено время работы параллельной программы на 2, 4, 8 и 16 процессах.
6. Было сделано профилирование на 16 процессах.

Оценка производительности (см. Приложение 3)

Показатели	Число процессов				
	1	2	4	8	16
Время, сек	67	54.79	32.42	17.28	11.48
Ускорение	1	1.23	2.07	3.87	5.83
Эффективность, %	100	61.15	51.66	48.45	36.46

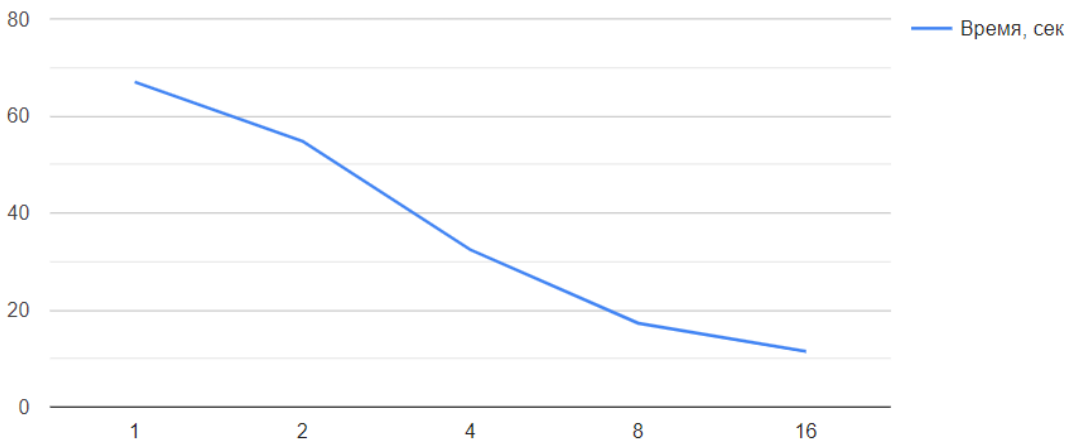


Рис.2 График времени

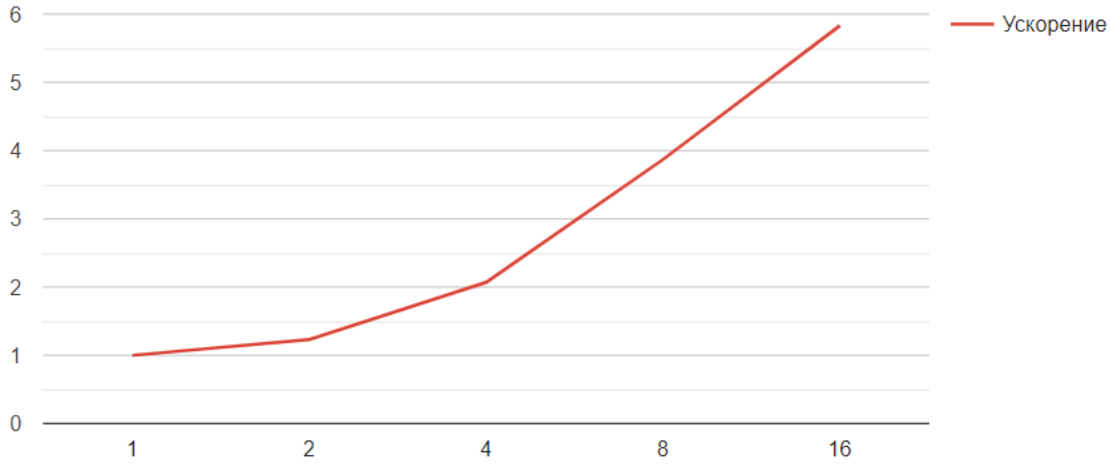


Рис.3 График ускорения

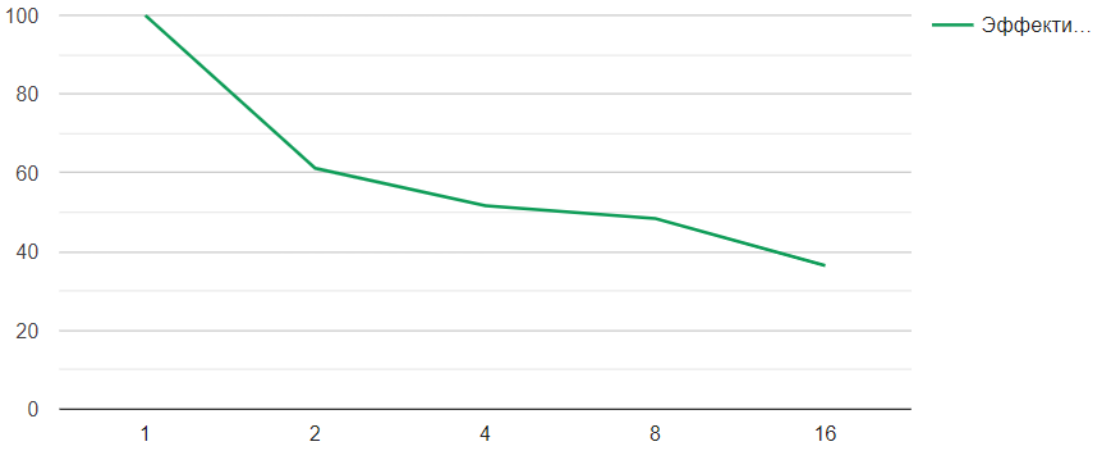
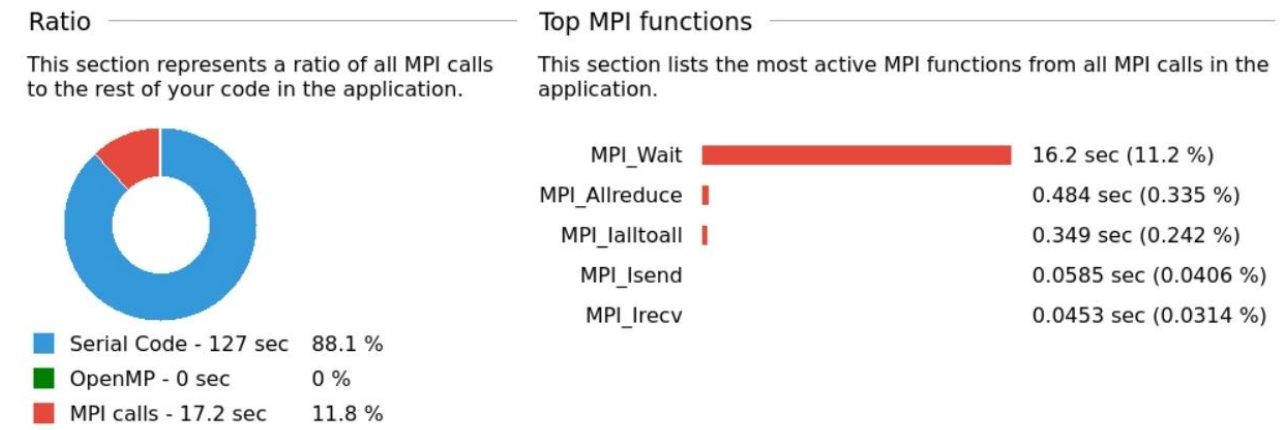
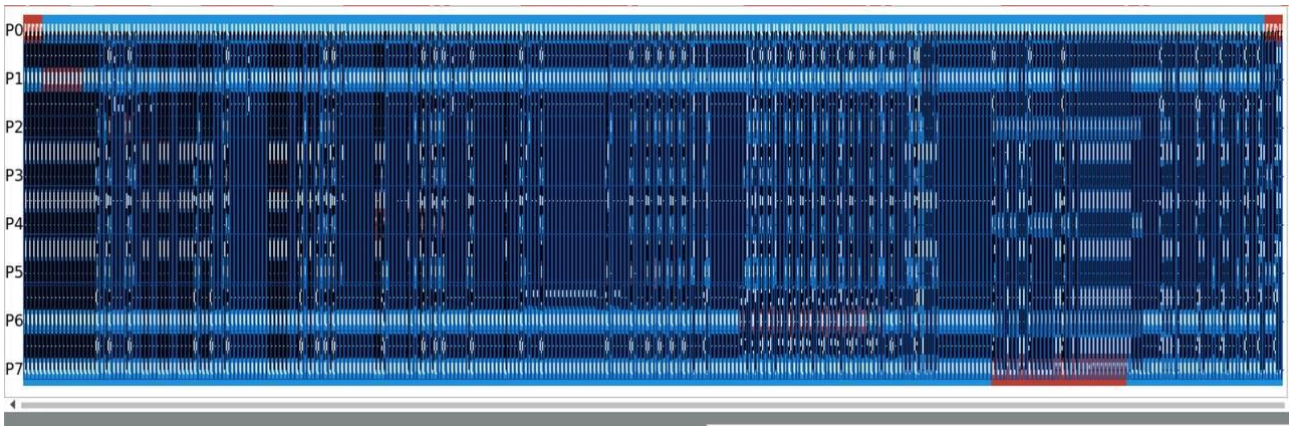


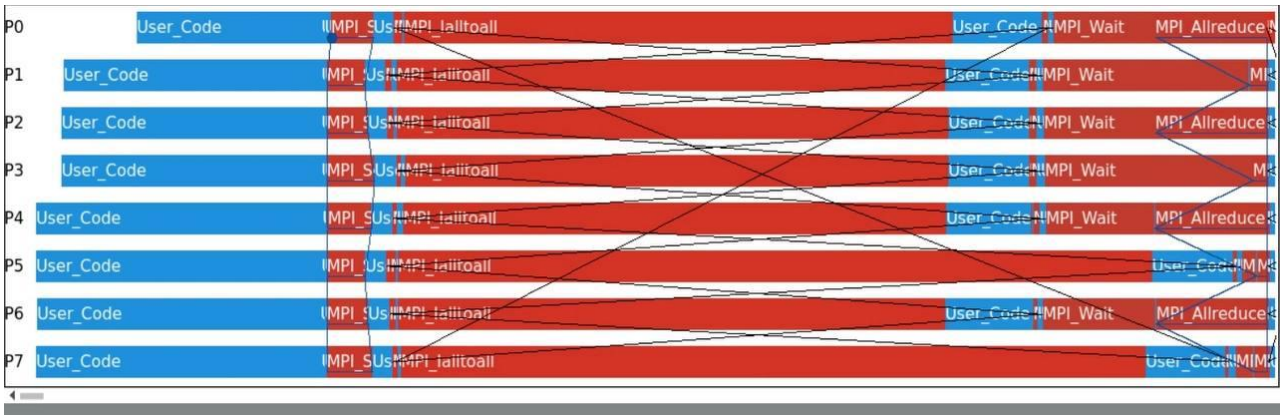
Рис. 4 График эффективности

Профилирование

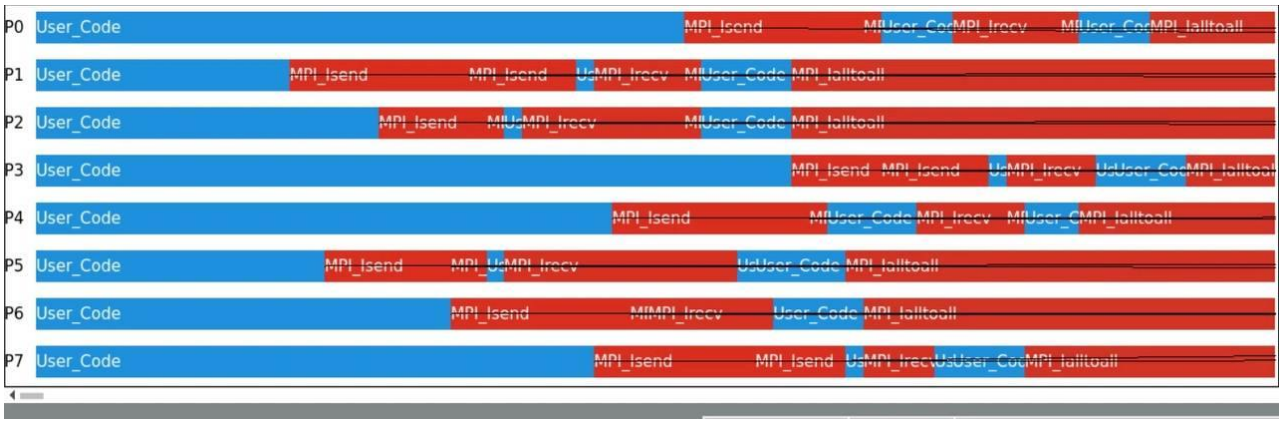
Было сделано профилирование на 8 MPI процессах, с использованием кафедрального сервера.

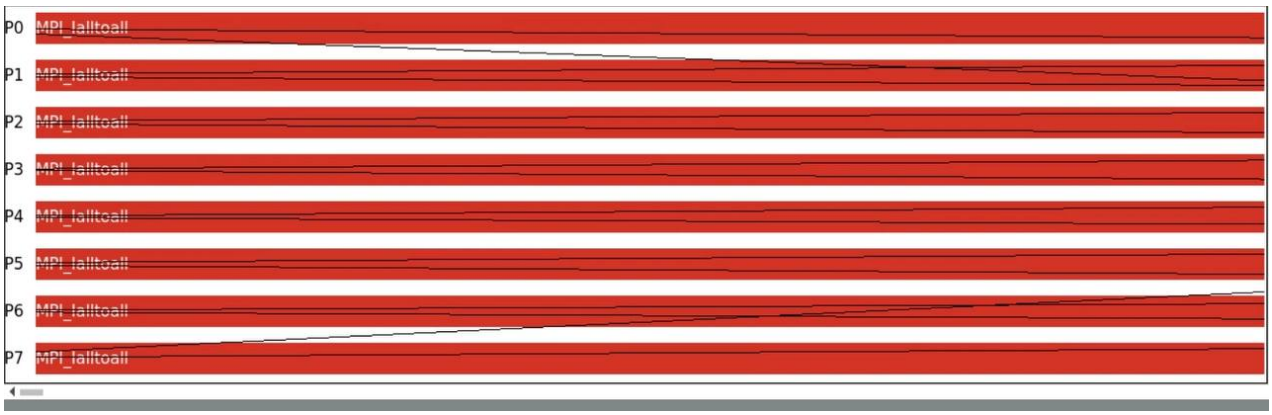


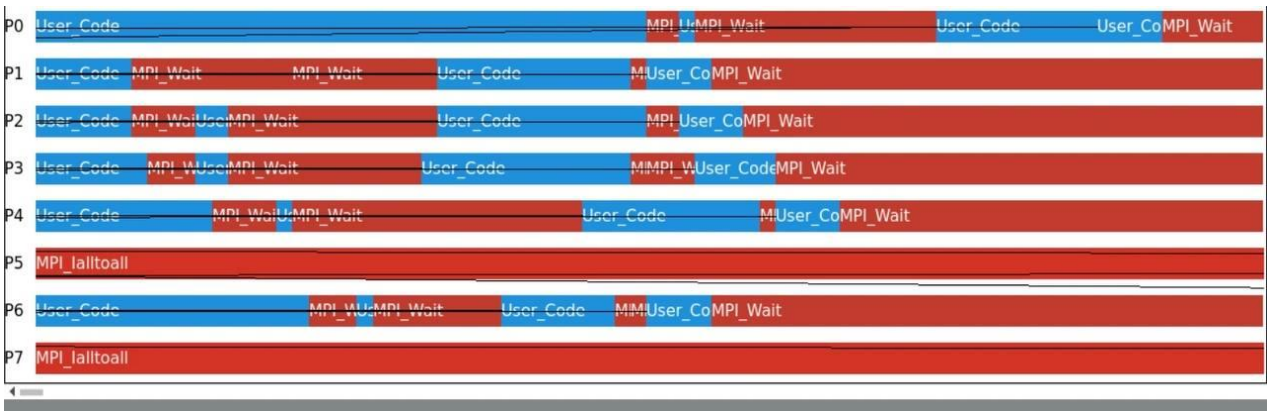


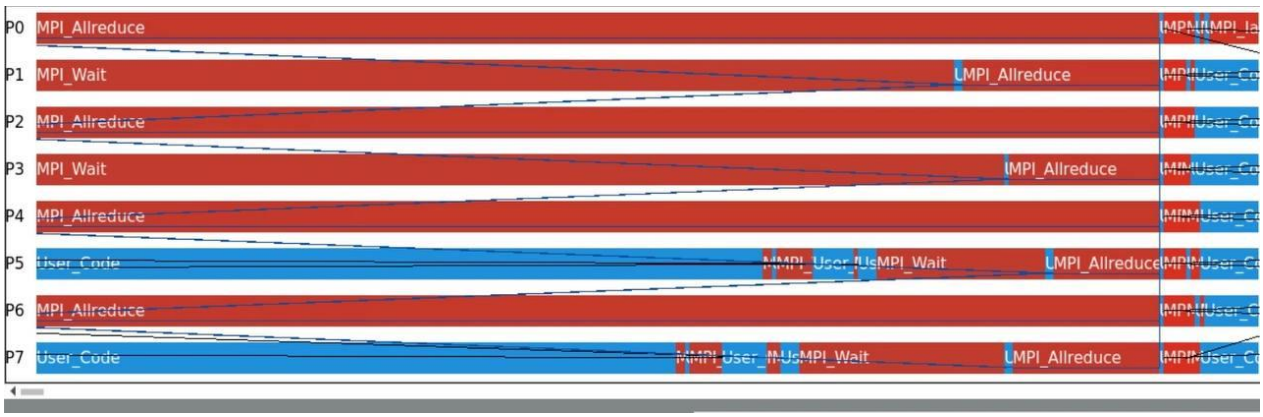



P0
P1
P2
P3
P4
P5
P6
P7
User_Code
MPI_Comm_size
MPI_Scatterv


P0
P1
P2
P3
P4
P5
P6
P7
User_Code
MPI_Isend
MPI_Irecv
MPI_Ialltoall


P0
P1
P2
P3
P4
P5
P6
P7
MPI_Ialltoall


P0
P1
P2
P3
P4
P5
P6
P7
User_Code
MPI_Wait
MPI_Ialltoall


P0
MPI_Allreduce
P1
MPI_Wait
MPI_Allreduce
P2
MPI_Allreduce
P3
MPI_Wait
MPI_Allreduce
P4
MPI_Allreduce
P5
MPI_Wait
MPI_Allreduce
P6
MPI_Allreduce
P7
User_Code
MPI_Wait
MPI_Allreduce

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы были освоены неблокирующие операции в MPI. С их помощью была распараллелена «Игра «Жизнь» Дж. Конвея» и удалось добиться ускорения в 5.8 раз.

Приложение 1. Листинг последовательной программы

```
#include <stdlib.h>
#include <iostream>

using namespace std;

#define NUM_ITER 50000

char* CreateFieldWithGlider(int numCols, int numRows) {
	char* field = (char*)calloc(numCols * numRows, sizeof(char));
	field[0 * numCols + 1] = 1;
	field[1 * numCols + 2] = 1;
	field[2 * numCols + 0] = 1;
	field[2 * numCols + 1] = 1;
	field[2 * numCols + 2] = 1;
	return field;
}

bool IsMatchFound(char** prevFields, char* curField, int sizeField, int numPassedIter) {
	for (int i = 0; i < numPassedIter; ++i) {
		bool isMatchDetected = true;
		for (int j = 0; j < sizeField; ++j) {
			if (prevFields[i][j] != curField[j]) {
				isMatchDetected = false;
				break;
			}
		}
		if (isMatchDetected) return true;
	}
	return false;
}

int GetNumOfAliveNeighborsInCentre(char* curPartOfField, int x, int y, int numRowsInCore,
int numCols) {
	int numNeighbors = 0;

	int prevx = (x - 1 + numCols) % numCols;
	int nextx = (x + 1 + numCols) % numCols;

	int prevy = (y - 1 + numRowsInCore) % numRowsInCore;
	int nexty = (y + 1 + numRowsInCore) % numRowsInCore;

	numNeighbors += curPartOfField[prevy * numCols + prevx];
	numNeighbors += curPartOfField[prevy * numCols + x];
	numNeighbors += curPartOfField[prevy * numCols + nextx];

	numNeighbors += curPartOfField[y * numCols + prevx];
	numNeighbors += curPartOfField[y * numCols + nextx];

	numNeighbors += curPartOfField[nexty * numCols + prevx];
	numNeighbors += curPartOfField[nexty * numCols + x];
	numNeighbors += curPartOfField[nexty * numCols + nextx];

	return numNeighbors;
}

bool isAlive(char* curPartOfField, int x, int y, int numCols) {
	if (curPartOfField[x + y * numCols] == 1) return true;
	return false;
}

bool isSurvived(int numNeighbors) {
	if (numNeighbors == 2 || numNeighbors == 3) return true;
	return false;
```

```
}

bool isBorn(int numNeighbors) {
	if (numNeighbors == 3) return true;
	return false;
}

void CalcNextGeneration(char* newPartOfField, char* curPartOfField, int numRows, int
numCols) {
	int numNeighbors = 0;
	for (int y = 0; y < numRows; ++y) {
		for (int x = 0; x < numCols; ++x) {
			numNeighbors = GetNumOfAliveNeighborsInCentre(curPartOfField, x, y,
numRows, numCols);
			bool keepAlive = isAlive(curPartOfField, x, y, numCols) &&
isSurvived(numNeighbors);
			bool becomeAlive = !isAlive(curPartOfField, x, y, numCols) &&
isBorn(numNeighbors);
			newPartOfField[x + y * numCols] = (keepAlive || becomeAlive == true ? 1 :
0);
		}
	}
}

void StartSimulation(char* currField, int numRows, int numCols) {
	char** prevFields = (char**)malloc(NUM_ITER * sizeof(char*));

	int countIter;

	for (countIter = 1; countIter <= NUM_ITER; ++countIter) {
		int numPassedIter = countIter - 1;
		char* nextField = (char*)malloc(numRows * numCols * sizeof(char));
		CalcNextGeneration(nextField, currField, numRows, numCols);
		if (IsMatchFound(prevFields, currField, numRows * numCols, numPassedIter)) {
			--countIter;
			break;
		}
		prevFields[numPassedIter] = currField;
		currField = nextField;
	}

	printf("Total iterations: %d\n", countIter);

	for (int i = 0; i < countIter; ++i) {
		free(prevFields[i]);
	}
	free(prevFields);
}

int main(int argc, char** argv) {
	const int numRows = atoi(argv[1]);
	const int numCols = atoi(argv[2]);

	char* field = CreateFieldWithGlider(numCols, numRows);

	time_t startTime;
	time(&startTime);

	StartSimulation(field, numRows, numCols);

	time_t endTime;
	time(&endTime);

	double totalTime = difftime(endTime, startTime);
	printf("Total time: %f sec\n", totalTime);
```

```
}
```

Приложение 2. Листинг параллельной программы

```
#include "mpi.h"
#include <stdlib.h>
#include <iostream>
#include <math.h>

using namespace std;

#define NUM_ITER 50000
#define T1 67.0

void PrintField(char* field, int numRows, int numCols) {
	for (int i = 0; i < numRows; ++i) {
		for (int j = 0; j < numCols; ++j) {
			std::cout << (field[i * numCols + j] == 1 ? "#" : ".") << " ";
		}
		std::cout << std::endl;
	}
	std::cout << std::endl;
}

char* CreateFieldWithGlider(int numCols, int numRows) {
	char* field = (char*)calloc(numCols * numRows, sizeof(char));
	field[0 * numCols + 1] = 1;
	field[1 * numCols + 2] = 1;
	field[2 * numCols + 0] = 1;
	field[2 * numCols + 1] = 1;
	field[2 * numCols + 2] = 1;
	return field;
}

int* CalcSizesOfParts(int numProc, int numRows, int numCols) {
	int* sizesOfParts = (int*)malloc(numProc * sizeof(int));
	double step = (double)numRows / numProc;
	double countLines = step;
	int numLinesOfPart = 0;
	int countBusyLines = 0;
	for (int i = 0; i < numProc; ++i) {
		numLinesOfPart = ceil(countLines) - countBusyLines;
		sizesOfParts[i] = numLinesOfPart * numCols;
		countBusyLines += numLinesOfPart;
		countLines += step;
	}
	return sizesOfParts;
}

int* CalcDisplsOfParts(int numProc, int* sizesOfParts) {
	int* displsOfParts = (int*)malloc(numProc * sizeof(int));
	int curOffset = 0;
	for (int i = 0; i < numProc; ++i) {
		displsOfParts[i] = curOffset;
		curOffset += sizesOfParts[i];
	}
	return displsOfParts;
}
```

```
void CalcVectorOfStopFlags(char* stopflags, char** prevPartOfFields, char*
     curPartOfField, int sizeOfPart, int curIter) {
     int numFlags = curIter - 1;
     for (int i = 0; i < numFlags; ++i) {
           stopflags[i] = 1;
           for (int j = 0; j < sizeOfPart; ++j) {
                 if (prevPartOfFields[i][j] != curPartOfField[j]) {
                       stopflags[i] = 0;
                       break;
                 }
           }
     }
}

int GetNumOfAliveNeighborsInCentre(char* curPartOfField, int x, int y, int
     numRowsInCore, int numCols) {
     int numNeighbors = 0;

     int prevx = (x - 1 + numCols) % numCols;
     int nextx = (x + 1 + numCols) % numCols;

     int prevy = (y - 1 + numRowsInCore) % numRowsInCore;
     int nexty = (y + 1 + numRowsInCore) % numRowsInCore;

     numNeighbors += curPartOfField[prevy * numCols + prevx];
     numNeighbors += curPartOfField[prevy * numCols + x];
     numNeighbors += curPartOfField[prevy * numCols + nextx];

     numNeighbors += curPartOfField[y * numCols + prevx];
     numNeighbors += curPartOfField[y * numCols + nextx];

     numNeighbors += curPartOfField[nexty * numCols + prevx];
     numNeighbors += curPartOfField[nexty * numCols + x];
     numNeighbors += curPartOfField[nexty * numCols + nextx];

     return numNeighbors;
}

bool isAlive(char* curPartOfField, int x, int y, int numCols) {
     if (curPartOfField[x + y * numCols] == 1) return true;
     return false;
}

bool isSurvived(int numNeighbors) {
     if (numNeighbors == 2 || numNeighbors == 3) return true;
     return false;
}

bool isBorn(int numNeighbors) {
     if (numNeighbors == 3) return true;
     return false;
}

void CalcNextGenerationInCentre(char* newPartOfField, char* curPartOfField, int
     numRowsInCore, int numCols) {
     int numNeighbors = 0;
     for (int y = 1; y < numRowsInCore - 1; ++y) {
           for (int x = 0; x < numCols; ++x) {
                 numNeighbors = GetNumOfAliveNeighborsInCentre(curPartOfField, x, y,
     numRowsInCore, numCols);
                 bool keepAlive = isAlive(curPartOfField, x, y, numCols) &&
     isSurvived(numNeighbors);
```

```
                bool becomeAlive = !isAlive(curPartOfField, x, y, numCols) &&
    isBorn(numNeighbors);
                newPartOfField[x + y * numCols] = (keepAlive || becomeAlive == true ?
    1 : 0);
            }
    }
}

int GetNumOfAliveNeighborsInRow(char* curPartOfField, char* recvRow, int x, int y, int
    numCols, int numRowsInCore, int rank) {
    int numNeighbors = 0;

    int prevx = (x - 1 + numCols) % numCols;
    int nextx = (x + 1 + numCols) % numCols;

    numNeighbors += curPartOfField[y * numCols + prevx];
    numNeighbors += curPartOfField[y * numCols + nextx];

    numNeighbors += recvRow[prevx];
    numNeighbors += recvRow[x];
    numNeighbors += recvRow[nextx];

    if (y == 0) {
        numNeighbors += curPartOfField[1 * numCols + prevx];
        numNeighbors += curPartOfField[1 * numCols + x];
        numNeighbors += curPartOfField[1 * numCols + nextx];
    }
    else {
        numNeighbors += curPartOfField[(y - 1) * numCols + prevx];
        numNeighbors += curPartOfField[(y - 1) * numCols + x];
        numNeighbors += curPartOfField[(y - 1) * numCols + nextx];
    }

    return numNeighbors;
}

void CalcNextGenerationInRow(char* newPartOfField, char* curPartOfField, char* recvRow,
    int y, int numRowsInCore, int numCols, int rank) {
    int numNeighbors = 0;
    for (int x = 0; x < numCols; ++x) {
        numNeighbors = GetNumOfAliveNeighborsInRow(curPartOfField, recvRow, x, y,
    numCols, numRowsInCore, rank);
        bool keepAlive = isAlive(curPartOfField, x, y, numCols) &&
    isSurvived(numNeighbors);
        bool becomeAlive = !isAlive(curPartOfField, x, y, numCols) &&
    isBorn(numNeighbors);
        newPartOfField[x + y * numCols] = (keepAlive || becomeAlive == true ? 1 :
    0);
    }
}

bool IsMatchFound(char* exchStopFlags, int sizeOfIterBlock, int numPassedIter, int
    numProc) {
    int flagOfMatch = 0;
    for (int i = 0; i < sizeOfIterBlock; ++i) {
        int countOnes = 0;
        for (int j = 0; j < numProc; ++j) {
            countOnes += exchStopFlags[i + j * sizeOfIterBlock];
        }
        if (countOnes == numProc) {
            flagOfMatch = 1;
            break;
```

```
            }
        }
        int numMatches = 0;
        MPI_Allreduce(&flagOfMatch, &numMatches, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

        if (numMatches >= 1) return true;
        return false;
}

void StartSimulation(char* curPartOfField, int numRows, int numCols, int* sizesOfParts,
        int numProc, int curcore, char* field, int* displs) {
        char** prevPartOfFields = (char**)malloc(NUM_ITER * sizeof(char*));

        char* stopFlags = (char*)calloc(NUM_ITER, sizeof(char));
        char* exchStopFlags = (char*)calloc(NUM_ITER, sizeof(char));

        int numRowsInCore = sizesOfParts[curcore] / numCols;

        int prevCore = (curcore == 0 ? numProc - 1 : curcore - 1);
        int nextCore = (curcore == numProc - 1 ? 0 : curcore + 1);

        char* lineFromPrevCore = (char*)malloc(numCols * sizeof(char));
        char* lineFromNextCore = (char*)malloc(numCols * sizeof(char));

        MPI_Request reqSendPrev;
        MPI_Request reqSendNext;
        MPI_Request reqRecvPrev;
        MPI_Request reqRecvNext;
        MPI_Request reqFlags;

        int countIter;

        for (countIter = 1; countIter <= NUM_ITER; ++countIter) {

                int numPassedIter = countIter - 1;

                MPI_Isend(curPartOfField, numCols, MPI_CHAR, prevCore, 0, MPI_COMM_WORLD,
        &reqSendPrev);

                int shiftToLastLine = sizesOfParts[curcore] - numCols;
                MPI_Isend(curPartOfField + shiftToLastLine, numCols, MPI_CHAR, nextCore, 0,
        MPI_COMM_WORLD, &reqSendNext);

                MPI_Irecv(lineFromPrevCore, numCols, MPI_CHAR, prevCore, MPI_ANY_TAG,
        MPI_COMM_WORLD, &reqRecvPrev);

                MPI_Irecv(lineFromNextCore, numCols, MPI_CHAR, nextCore, MPI_ANY_TAG,
        MPI_COMM_WORLD, &reqRecvNext);

                CalcVectorOfStopFlags(stopFlags, prevPartOfFields, curPartOfField,
        sizesOfParts[curcore], countIter);

                int sizeOfIterBlock = ceil((double)numPassedIter / numProc);
                MPI_Ialltoall(stopFlags, sizeOfIterBlock, MPI_CHAR, exchStopFlags,
        sizeOfIterBlock, MPI_CHAR, MPI_COMM_WORLD, &reqFlags);

                char* newPartOfField = (char*)malloc(sizesOfParts[curcore] * sizeof(char));

                CalcNextGenerationInCentre(newPartOfField, curPartOfField, numRowsInCore,
        numCols);

                MPI_Wait(&reqSendPrev, MPI_STATUS_IGNORE);
```

```
            MPI_Wait(&reqRecvPrev, MPI_STATUS_IGNORE);

            CalcNextGenerationInRow(newPartOfField, curPartOfField, lineFromPrevCore, 0,
      numRowsInCore, numCols, curcore);

            MPI_Wait(&reqSendNext, MPI_STATUS_IGNORE);

            MPI_Wait(&reqRecvNext, MPI_STATUS_IGNORE);

            CalcNextGenerationInRow(newPartOfField, curPartOfField, lineFromNextCore,
      numRowsInCore - 1, numRowsInCore, numCols, curcore);

            MPI_Wait(&reqFlags, MPI_STATUS_IGNORE);

            if (IsMatchFound(exchStopFlags, sizeOfIterBlock, numPassedIter, numProc)) {
                  --countIter;
                  break;
            }
            prevPartOfFields[numPassedIter] = curPartOfField;
            curPartOfField = newPartOfField;
      }

      if (curcore == 0) {
            cout << "Total iterations: " << countIter << endl;
      }

      for (int i = 0; i < countIter; ++i) {
            free(prevPartOfFields[i]);
      }
      free(prevPartOfFields);
      free(stopFlags);
      free(exchStopFlags);
      free(lineFromNextCore);
      free(lineFromPrevCore);
}

void PrintResult(float totalTime, int numProc) {
      float boost = T1 / totalTime;
      float efficiency = (boost / (float)numProc) * 100;
      printf("Number of processes: %d\n", numProc);
      printf("Total time: %f sec\n", totalTime);
      printf("Sp = %f\n", boost);
      printf("Ep = %f\n\n", efficiency);
}

int main(int argc, char** argv) {
      int numProc;
      int rank;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &numProc);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

      char* field = NULL;
      const int numRows = atoi(argv[1]);
      const int numCols = atoi(argv[2]);

      if (rank == 0) {
            field = CreateFieldWithGlider(numCols, numRows);
      }
```

```
    double startTime = MPI_Wtime();

    int* sizesOfParts = CalcSizesOfParts(numProc, numRows, numCols);
    int* displsOfParts = CalcDisplsOfParts(numProc, sizesOfParts);

    char* partOfField = (char*)malloc(sizesOfParts[rank] * sizeof(char));

    MPI_Scatterv(field, sizesOfParts, displsOfParts, MPI_CHAR, partOfField,
    sizesOfParts[rank], MPI_CHAR, 0, MPI_COMM_WORLD);

    StartSimulation(partOfField, numRows, numCols, sizesOfParts, numProc, rank, field,
    displsOfParts);

    double endTime = MPI_Wtime();

    if (rank == 0) {
          PrintResult(endTime - startTime, numProc);
          free(field);
    }
    free(sizesOfParts);
    free(displsOfParts);

    MPI_Finalize();
}
```

Приложение 3. default.sh

```
default.sh          [-M--]  0 L:[  1+12  13/ 13] *(198 / 19
#!/bin/bash

#PBS -l walltime=00:20:00
#PBS -l select=1:ncpus=1:mpiprocs=1:mem=4000m,place=free
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')

./default 400 400.
```

Приложение 4. parallel.sh

```
parallel.sh         [----] 59 L:[  1+13  14/ 14] *(414 / 414b) <EOF>
#!/bin/bash

#PBS -l walltime=00:20:00
#PBS -l select=2:ncpus=8:mpiprocs=8:mem=8000m,place=free
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')

mpirun -machinefile $PBS_NODEFILE -np 2 ./parallel 400 400
mpirun -machinefile $PBS_NODEFILE -np 4 ./parallel 400 400
mpirun -machinefile $PBS_NODEFILE -np 8 ./parallel 400 400
mpirun -machinefile $PBS_NODEFILE -np 16 ./parallel 400 400
```

Приложение 5. Результат работы последовательной программы

```
hpcuser60@clu:~/lab4> cat default.sh.o5513256
Total iterations: 1600
Total time: 67.000000 sec
hpcuser60@clu:~/lab4>
```

Приложение 6. Результат работы параллельной программы

```
hpcuser60@clu:~/lab4> cat parallel.sh.o5513268
Total iterations: 1600
Number of processes: 2
Total time: 54.786259 sec
Boost: 1.222934
Efficiency: 61.146717

Total iterations: 1600
Number of processes: 4
Total time: 32.425121 sec
Boost: 2.066299
Efficiency: 51.657478

Total iterations: 1600
Number of processes: 8
Total time: 17.284409 sec
Boost: 3.876326
Efficiency: 48.454075

Total iterations: 1600
Number of processes: 16
Total time: 11.486177 sec
Boost: 5.833098
Efficiency: 36.456863

hpcuser60@clu:~/lab4>
```